9 Middleware for Process Coordination

Software agents provide a promising alternatives for Internet workflow management. Agents can manage resources, act as case managers, brokers, matchmakers, monitor the environment, or serve as enactment engines.

In this chapter we dissect an agent-based framework capable to support process coordination. First, we discuss the message-oriented distributed object system, then we introduce the component-based architecture for building agents.

Our thinking and design choices were influenced by existing systems and, whenever possible, we adopted ideas and integrated implementations fitting our agent model. We integrated with relative ease JESS, a Java Expert System Shell from Sandia National Laboratory, [20], and a tuple space, the `TSpaces` from IBM, [28].

At the time of this writing the Bond system consists of about $100,000$ lines of Java code and 700 Java classes grouped into `core,` `agents`, and `applications` packages.

Throughout this chapter we discuss our basic design philosophy, introduce the concepts, and discuss the implementation of various components. Whenever necessary we list the relevant source code or the pseudo-code and comment the functions provided. To fully understand the system the reader needs to examine the actual source code.

9.1 BOND CORE

In our view an agent is an active mobile object with some level of intelligence. *Active* means that the object has one or more threads of control, *mobile* means that the object may migrate from one site to another, *intelligence* means that the object has some degree of learning, planning, and/or inference capabilities. From this definition of agents it follows that we need first to construct an infrastructure for distributed objects and then build an agent framework on top of this infrastructure.

A basic design choice for a distributed object systems is the communication paradigm, a system may use remote method invocation, or message passing or possibly both. The two paradigms are dual, the same functionality that can be achieved with one of them, can be provided by the other. Systems supporting asynchronous communication typically use message passing while those supporting synchronous communication use remote method invocation.

Several general-purpose distributed-object systems are based upon remote method invocation, e.g., implementations of CORBA [42], like Visibroker [53] from Inprise or Orbix [43], from IONA, Java RMI [48] or Microsoft's DCOM. There are also few message-oriented distributed systems, like MSMQ from Microsoft, or iBUS [29] from SoftWired.

The `bond.core` package implements a message-oriented distributed object system, [2]. This section covers Bond objects, communication, and message handling.

9.1.1 Bond Objects

A Bond program is a flat collection of Bond objects. A Bond object extends the standard Java object with:

(i) *A unique identifier:* every Bond object is assigned a unique identifier for the lifetime of the object. An entire collection of Bond objects can be identified by an *alias*.

(ii) *Dynamic properties:* Bond objects may have dynamic properties created at runtime, in addition to the regular fields of a Java object.

(iii) *Communication support:* all Bond object are capable to receive messages.

(iv) *Registration with a local directory:* Bond objects are registered at the creation time with a local directory. They can be found using either the unique identifier or an alias. Lightweight Bond objects are registered on demand.

(v) *Serialization and cloning:* all Bond object are serializable and clonable, while only some Java objects are. The serialization and cloning functions are overwritten to accommodate the unique id of Bond objects.

(vi) *Multiple inheritance:* the Bond system extends the Java object model with multiple inheritance using a preprocessor of Java files.

(vii) *A visual editor:* all Bond objects can be visually edited.

We now discuss basic properties of Bond objects.

9.1.1.1 Bond Identifiers.

Every Bond object has a unique identifier, `bondID` generated by its constructor as follows:

```
bondID= ``bondID'' +  bondIPaddress + commEnginePort +
      localMillisecondSinceStartOfResident + timeAndDate
```

Here “+” stands for string concatenation, "bondID" is a string, `bondIPaddress` is the IP address of the host where the Bond system is running, `commEnginePort` is the port number of the Bond Communication Engine, the next string gives the time in milliseconds since the local Bond Resident was started, and `timeAndDate` is a string giving the hour, minute, second, day, month and year when the object was created. The resident and the communication engine are discussed in Sections 9.1.1.2 and 9.1.2.7, respectively.

This algorithm is fast and guarantees the uniqueness of the `bondID`.

The `bondID` remains the same throughout the lifetime of an object, it is invariant to operations such as: saving and loading the object to/from persistent storage or transferring to/from remote locations.

In Bond we have a *flat namespace*, the `bondID` does not carry information about the type or role of the object. A flat name-space cannot be used for routing during communication, or for classifying objects a useful feature for directory services. While these difficulties are real, they are inherent to the problem, not to the naming scheme.

A hierarchical naming scheme like IP-addresses cannot be used for a distributed object system supporting mobility because the id of the object should be the same after migration. On the other hand, most Bond objects remain at their creation site, thus the host information, contained in the `bondID` can be used as a way of speeding up the search. In this case, when the object is not found at the resident sub-field in its id, a global directory search is carried out.

Bond is a message-oriented system and each object identified by its unique `bondID` can receive messages. The `say()` method discussed in Section 9.1.3.4 is used to deliver a message to an object.

9.1.1.2 Bond Resident. A Bond resident, `bondResident` is a container object hosting all Bond objects located within a given virtual machine. Every Bond resident contains a *local directory*, `bondDir`, implemented as a singleton object [18] and a Communication Engine, `bondCommEngine`, see Section 9.1.2.7.

The constructors for a Bond executable and for a resident are:

```
public class bondExecutable extends bondObject
                              implements Runnable {
  public bondExecutable() {}
  public bondExecutable(boolean reg) { super(reg); }
  public void run(){} }

public class bondResident extends bondExecutable {
  public bondResident() {dir.addAlias("Resident", this);}
}
```

Other objects are loaded dynamically as needed, using dynamic probes, presented in Section 9.1.3.3. For example, whenever a message for an object called an Agent Factory arrives, the object is loaded and the message is passed to it. Agent Factories are discussed in Section 9.2.

A resident can be configured as a client, some type of server, e.g. Authentication, Persistent Storage, Directory Server, or as host for a number of agents.

The procedure called to initialize the Bond system is:

```
public static void initbond() {
  bondConfiguration.initSysProperties();
  loader = new bondLoader();
  dir = new bondDirectory();
  com = new bondCommunicator();
  conf = new bondConfiguration();
  bondMessage.initMessage();
  return;
}
```

A configuration file specifies the options requested by the user and a resident is configured accordingly. Section 9.1.3.6 addresses this issue in more detail.

9.1.1.3 Local Directory and Aliases.

Bond objects are registered automatically with the local directory at the creation time. Objects loaded from persistent storage and objects arriving from a remote location as a result of a `realize()` operation are registered at their instantiation time. Registration with the local directory is a precondition for any object to receive messages.

Lightweight objects are the only exception to the automatic registration rule. There are two classes of lightweight objects: `bondShadow` discussed in Section 9.1.1.5 and `bondMessage`. Messages and shadows have a unique `bondID` and can be registered with the local directory, if needed.

Registered objects cannot be *freed* in the Java sense because the local directory keeps a pointer to them. Thus, they are not garbage collected, until *unregistered*. Unregistering an object removes its ability to receive messages and makes it eligible to be garbage collected by Java. To un-register a Bond object `bo`: `dir.unregister(bo)`.

An object can be registered using either its unique `bondID` or an *alias*. An object may have multiple aliases and multiple objects may have the same alias. Objects with the same alias form an equivalence class and are indistinguishable from one another at some level. The alias mechanism implements the so called *anycast* addressing abstraction.

For example, within the same resident we may have several agent factories. A user or an agent who needs to create or migrate an agent at/to that site, sends a message with the alias "AgentFactory" as destination object. Upon receipt of the message, the local directory selects one of the objects in the class at random, and delivers the message to it. In its reply, the selected agent factory responds with its unique `bondID`. The addressing ambiguity is resolved after the first message exchange and subsequent communication carries the unique identifier of the object.

The alias system supports *load balancing* for servers. If multiple servers are registered under the same alias, an incoming request is delivered to a randomly chosen object, therefore dividing the load amongst servers. A server can even choose to temporarily un-register itself from the alias if it is overloaded, without affecting its current clients because they communicate with the server using its unique identifier.

Table 9.1 Reserved aliases

Alias	Function
Resident	Container object for all Bond objects at a given site.
AgentFactory	Create, destroy, checkpoint and migrate agents.
PSS	Persistent storage service: save/retrieve objects from storage.
Directory	Directory service: remote access interface for the local directory.
Monitor	Monitoring service.

Table 9.1 lists *reserved aliases* for standard Bond services.

9.1.1.4 Serialization and Cloning. *Serialization* allows an object to be saved in an input/output stream, it *flattens* the object. *Cloning* creates an exact copy of the object.

Java objects can be serialized if they implement the `Serializable` interface. This interface does not implement new methods. Threads are not serializable in Java.

All Bond objects are serializable and can be cloned, `bondObject`, the root of the object hierarchy, implements the `Serializable` interface and re-implements the `clone()` method. A clone of a Bond object is identical with the original object, but has a different `bondID`.

```
public class bondObject implements Serializable, Cloneable {
 public bondObject() {
    maybeInformDirectory();
    if (bondID != null) { setName(bondID); }
    }
  public bondObject(boolean val) {
    if (val) {
      bondID = dir.getBondID();
      setName(bondID); }
  }
  private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    if (bondID !=null)   dir.register(this);
  }
   protected void maybeInformDirectory() {
    try {
      bondID = dir.getBondID();
      dir.register(this);
    } catch (NullPointerException e)
   }
```

9.1.1.5 Bond Shadows. A distributed system needs an abstraction for communication with remote objects. In Voyager [21] this abstraction is called a *proxy*.

CORBA [42] and Java Remote Method Invocation, RMI [48] call it a *stub*. In Bond this abstraction is a lightweight object called a *shadow*.

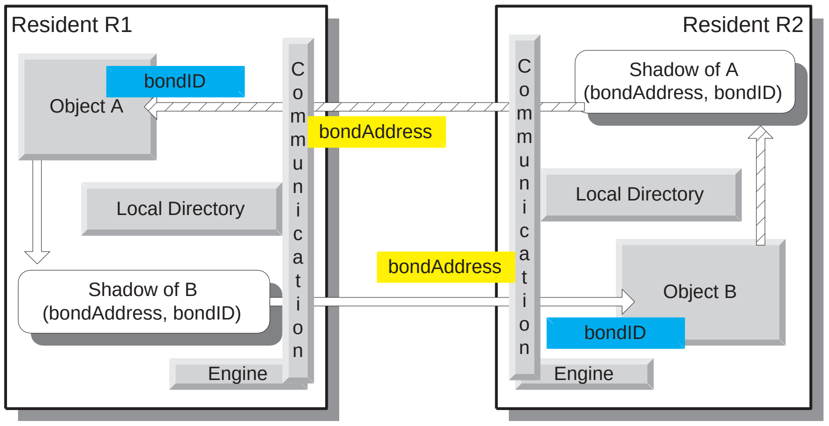


Fig. 9.1 Communication with remote objects. A Bond communication engine runs at a known port at on a host identified by an IP address. A `bondAddress` consists of a pair `(bondIPaddress,commEnginePort)`. In turn, a Bond shadow consists of the pair `(bondAddress, bondID)`. To send a message to a remote object B, object A needs a shadow of B. Once this shadow exists then A sends messages for B to its shadow.

To communicate with objects hosted by a `bondResident` we need to know the IP address of the host were the Bond system is running. The communication engine runs at a known port on that host. The pair (bondIPaddress,commEnginePort) defines a `bondAddress`.

In turn, a shadow of an object A consists of the pair `(bondAddress, bondID)`, where `bondAddress` allows us to uniquely identify a resident and `bondID` uniquely identifies the object on that resident.

In Bond there is no distinction between communication with local and with remote objects, a message delivered to the local shadow is guaranteed to reach the remote object. Moreover, the `realize()` method allows us to create a local copy of a remote object when we have a shadow of the object. The local copy created with the `realize()` method has the same `bondID` as the original object.

The constructor for a `bondShadow` and the `realize()` method supporting object migration are:

```
/**  Default constructor. Don't register.*/
 public bondShadow() {
     super(false);
  }
/** Create shadow from bondID and address of object*/
 public bondShadow(String remote_bondID,
                                String rem_address){
```

```
     super(false);
     this.remote_bondID= remote_bondID;
     remote_address = new bondIPAddress(rem_address);
  }
/**  Create shadow from an address */
 public bondShadow(String remote_bondID,
                  bondIPAddress address){
     super(false);
     this.remote_bondID= remote_bondID;
     remote_address = (bondIPAddress)address.clone();
  }
/**  Create shadow of a local object */
  public bondShadow(bondObject bo) {
      super(false);
      local = bo;
  }
/**  Object migration */
 public bondObject realize() {
      bondID = dir.getBondID();
      dir.register(this);
      bondObject bo = null;
      if (local != null) {
          return local;
      } else {
          bondMessage m =new bondMessage(
            "(tell :content realize)", "PropertyAccess");
          m.setNeedsReply();
          say(m, this);
          m.waitReply(30000);
          return m.bo;  }
  }
```

Figure 9.1 illustrates full-duplex communication between two objects A and B registered with residents, R1 and R2 on two different systems. A sends messages for B to the shadow of B on R1 and, in turn, B sends messages to A to the shadow of A on R2.

9.1.1.6 Dynamic Object Properties. The ability to create on demand new properties/fields of an object is a feature of programming languages like Lisp or Scheme that allow programmers to handle data whose name or type is not known at compile time. The compilers and linkers for programming languages like C or C++ usually discard the names of the variables, keeping only their address in the compiled code. Java, keeps this information in the compiled class files and allows access to it through a mechanism called *reflection*.

Dynamic properties are important for software agents, their functionality makes it difficult to anticipate all the fields of an agent at the instance the agent is created. For efficiency reasons regular Java fields should be used whenever possible, and we should resort to dynamic fields only when the name and/or type of the field is not known at compile time. Dynamic properties have a longer access time than regular Java fields, but for remote objects this difference is masked by the network latency. Compile-time type checking cannot be done for dynamic properties, thus the programmer looses important type-safety information.

Bond objects implement a common interface with two methods, `get` and `put`, to access static fields and dynamic properties:

`Object get(String name);` returns the value of the field or dynamic property called “name”. Numerical values, which are not objects in Java, are first converted to their object counterpart, e.g., an `int` is converted to an `Integer` object. The `get` function returns `null` when their is no object or field with the given name.

`Object set(String name, Object value)` sets the value of the field or dynamic property called `name` to the value specified by `value`. If there is no field or dynamic property with the given name, a new dynamic property is created. If there is a field with the given name but its type conflicts with the type of the object `value`, a casting exception is thrown.

All dynamic properties are considered to be of type `Object` and any value can be set for them. To delete a dynamic property, the `value` is set to `null`.

The `set` and `get` functions support multilevel access using the familiar dotted notation. Assume that we have a Bond object `foo` with `boo` as a field with `boo` a Bond object with a `name` field. The `set` and `get` functions applied to `foo` allows us to set the `name` field of its member object to the value “hector” and to retrieve the value:

```
foo.set(``boo.name'', ``hector'')}.
String val =foo.get(``boo.name'')}
```

The multilevel addressing can be done to arbitrary depths. This facility increases the access time due to overhead for parsing the string. Multilevel addressing can be turned off by setting the `useAccessors` boolean variable in the Bond configuration object.

The property access subprotocol discussed in Section 9.1.3.5 can be used to access the fields and dynamic properties without making a local copy of a potentially large remote object.

9.1.1.7 Multiple Inheritance.

Multiple inheritance is a controversial feature of object-oriented programming languages. Some, like C++ and Eiffel, support it, while others like Java, Objective C, and Modula-3 do not. Name resolution, repeated inheritance, and more obscure and difficult to read code are some of the problems associated with multiple inheritance.

Sometimes multiple inheritance is necessary because an object may have multiple roles. Even in the base Java classes `InputOutputStream` is a specialization of both `InputStream` and `OutputStream`. There are several examples of multiple inheritance in Bond, for example `bondMessage` may inherit KQML and XML parser classes.

Java allows *multiple interface inheritance*, but does not implement multiple class inheritance. There are *ad-hoc methods* to circumvent the limitations of Java, [52]:

(i) *Copy and modify:* copy the code otherwise inherited and modify it.

(ii) *Base class modification:* modify the base class to eliminate the need for multiple subclasses.

(iii) *Delegation:* create a member object and a number of wrapper functions which forward the requests to the member object.

The first two techniques require access to the source code and lead to code duplication and poor quality code. Our approach to multiple inheritance is similar with the one of the Jamie system at the University of Virginia, [52], it is based upon preprocessing into regular Java code but uses a less elaborate merging approach.

The source code is created in Bond from specially constructed files, with the `.bj` extension, instead of the regular `.java` extension. These files may contain variables and methods, but only one of them has the regular class headers. The `.bj` files are preprocessed by the `cpp` preprocessor of the GNU C/C++ distributions, to create the Java source code.

This implementation does not solve the problem of multiple sub-typing, or provide name resolution. The names of variables and methods should be disjoint. The method supports *conditional multiple inheritance*, the inheritance changes depending upon the configuration of the system.

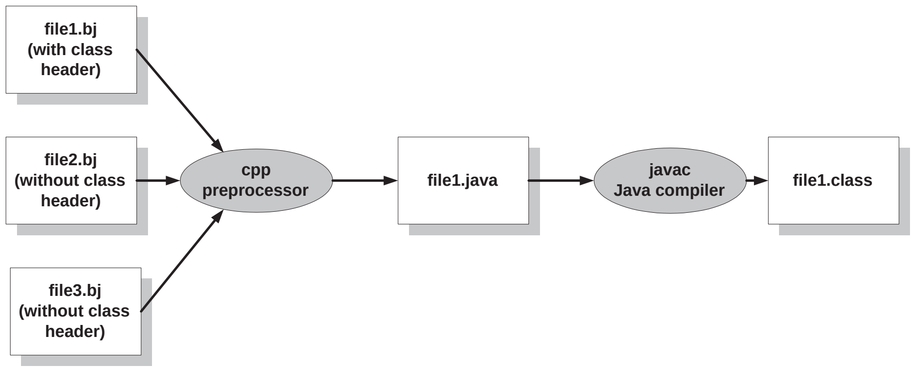
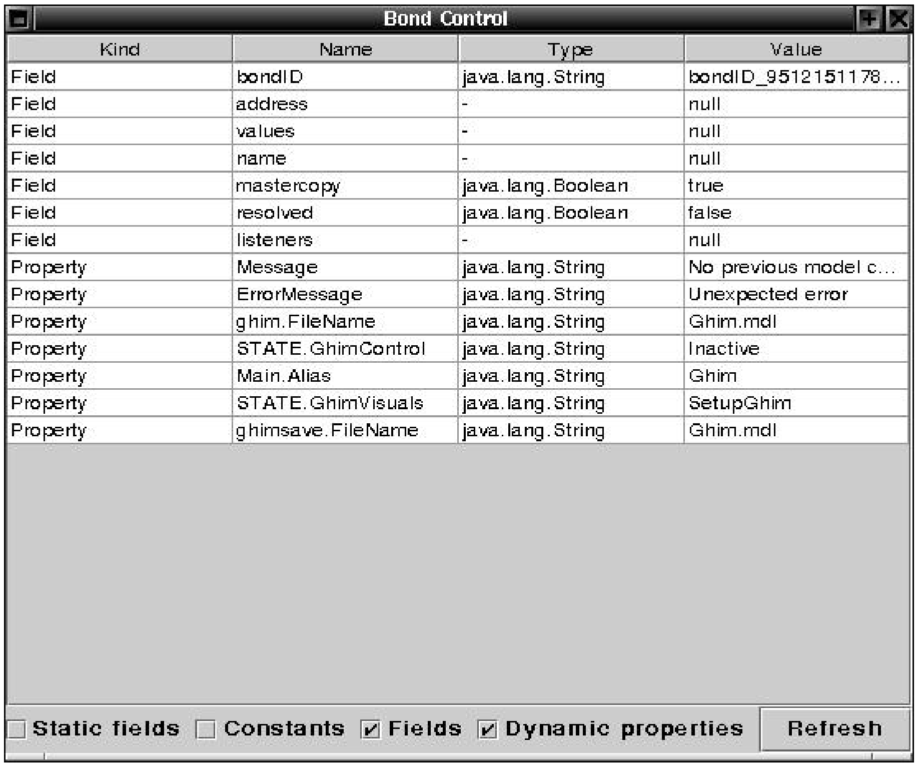

file1.bj (with class header)
file2.bj (without class header)
file3.bj (without class header)
cpp preprocessor
file1.java
javac Java compiler
file1.class

Fig. 9.2 Implementation of multiple inheritance in Bond

Only the original `.bj` files should be modified. The dependency from the `.bj` files to the Java file is expressed in the `makefile` of the Bond system, and the Java file are recreated whenever the `.bj` files are modified.

9.1.1.8 Visual Editing of Objects Bond objects can be visualized and edited. *Object editors* consist of independent dialog boxes and allow us to edit the fields and the dynamic properties of an object, as shown in Figure 9.3. They provide a functionality similar to the *property sheets* and *bean customizers* of the JavaBeans.

Visual objects represent an object and show its state and relationships with other objects as seen in Figure 9.4.



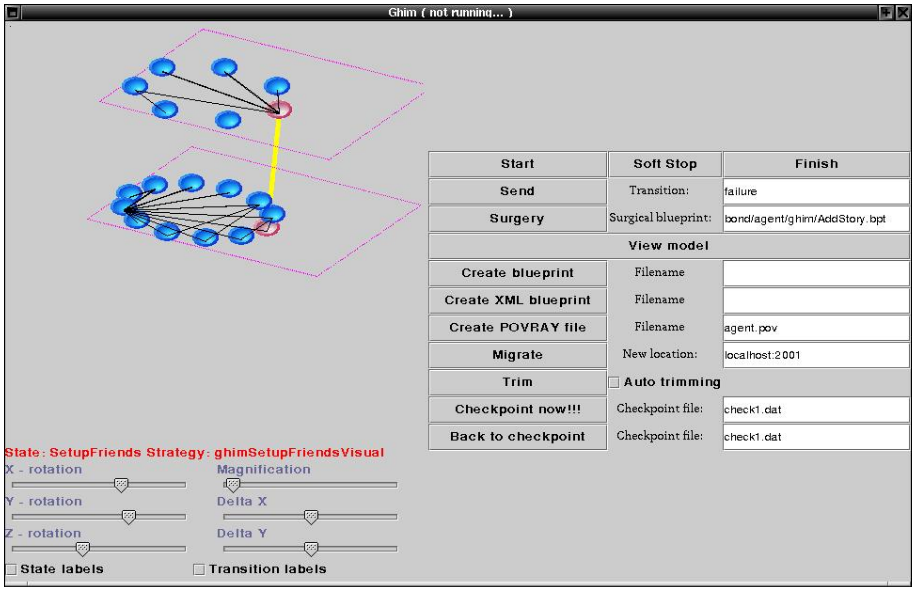
Kind	Name	Type	Value
Field	bondID	java.lang.String	bondID_9512151178...
Field	address	-	null
Field	values	-	null
Field	name	-	null
Field	mastercopy	java.lang.Boolean	true
Field	resolved	java.lang.Boolean	false
Field	listeners	-	null
Property	Message	java.lang.String	No previous model c...
Property	ErrorMessage	java.lang.String	Unexpected error
Property	ghim.FileName	java.lang.String	Ghim.mdl
Property	STATE.GhimControl	java.lang.String	Inactive
Property	Main.Alias	java.lang.String	Ghim
Property	STATE.GhimVisuals	java.lang.String	SetupGhim
Property	ghimsave.FileName	java.lang.String	Ghim.mdl

Fig. 9.3 Screenshot of the Bond object editor, displaying the content of the model of the agent.

Bond editors objects inherit from the `bondEditor` and are created on demand whenever the `edit()` function of the original object is called. The editor of an Bond object is itself an object attached to the `editor` dynamic property of the original object, and removed when the object is destroyed.

The mechanism described below ensures that every object can be edited and allows the user to customize the editor. The Bond editor object is accessed by *name lookup with inheritance based fallback* as follows: given an object of type `a.b.c` the system attempts to create an editor object of type `a.b.cEditor` and if that fails, an object of type `bond.core.editor.cEditor`. If this attempts fails, too, the system determines the first ancestor of the object and repeats the process recursively. The `bond.core.editor.bondObjectEditor` is invoked as the last resort as the editor for the object because every object inherits from `bondObject`.

Visual objects inherit from the `bondVisualObject` object and are attached to the `visual` dynamic property of the original object. Visual objects do not have a window of their own, instead they are represented as a graphic widget in the context of an editor presenting multiple objects and relations at the same time. An example is



State: SetupFriends Strategy: ghimSetupFriendsVisual

Transition:	failure
Surgical blueprint:	bond/agent/ghim/AddStory.bpt
Filename	
Filename	
Filename	agent.pov
New location:	localhost:2001
Checkpoint file:	check1.dat
Checkpoint file:	check1.dat

Fig. 9.4 Screenshot of the Bond agent editor. The bullets and lines represent the visual objects attached to Bond objects components the state machine.

the visual representation of the multi-plane finite state machine of the agents shown in Figure 9.4. The states and transitions have attached visual objects displayed on the screen, bullets for the states, lines for transitions. The visual representation reflects the internal state of the object.

9.1.1.9 Bond Loader. The `bondLoader` allows ordinary Bond objects to be loaded dynamically from the local repository using a search path constructed statically. It also allows loading of a class given the `URL` of a remote repository. The skeleton of the `bondLoader` code follows.

```
public class bondLoader extends bondObject {
  public Vector defaultpath = null;
  ClassLoader cloader = null;

  public bondLoader() {
   // default loading path
   defaultpath = new Vector();
   defaultpath.addElement("bond.core.");
   defaultpath.addElement("bond.services.");
   defaultpath.addElement("bond.agent.");
   defaultpath.addElement("bond.application.");
   defaultpath.addElement("bond.application.TupleSpace.");
  }
```

```
/** Create object "name" with default constructor
    given "searchpath" */
 public Object load(String name, Vector searchpaths) {
      Object o = null;
      Class cl;
      try {
      if ((cl = loadClass(name, searchpaths)) != null) {
          o = cl.newInstance(); }
      } catch(IllegalAccessException cnfe) { }
        catch(InstantiationException ie) { }
      return o;
  }
/** Load a local class given its "name" */
 public Class loadClass(String name, Vector searchpaths) {
      String completename;
      for(int i=0; i!=searchpaths.size(); i++) {
        try {
          Class cl = Class.forName(makeName
             (name, (String)searchpaths.elementAt(i)));
          return cl;
        } catch(ClassNotFoundException cnfe) {
        } catch(Exception ex) { }
      }
/** Load classes remotely */
  if (cloader == null) {
    String c_repository = System.getProperty
                     ("bond.current.strategy.repository");
    String repository = System.getProperty
                     ("bond.strategy.repository");
    if (c_repository != null && repository != null) {
      try {
        URL urlList[] = {new URL(c_repository),
                                  new URL(repository)};
        cloader = new URLClassLoader(urlList);
      }
      catch (MalformedURLException e) { }
    }
    else if (repository != null) {
      try {
        URL urlList[] = {new URL (repository)};
        cloader = new URLClassLoader(urlList);
      }
      catch (MalformedURLException e) { }
    }
   }
    for (int i = 0; i < searchpaths.size(); i++) {
      try {
          Class cl = Class.forName(makeName(name,
                   (String)searchpaths.elementAt(i)),
```

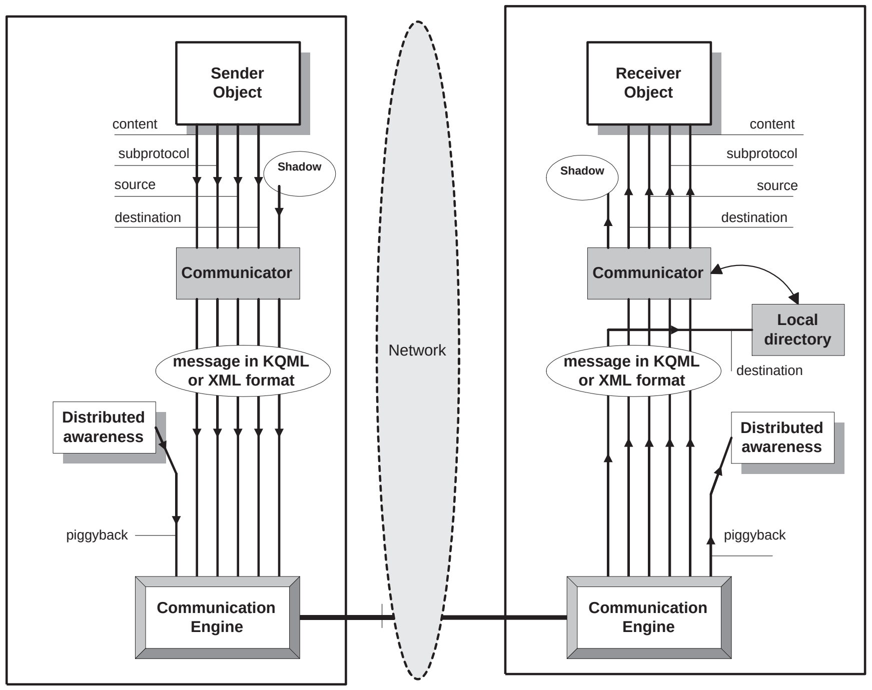


Fig. 9.5 Message delivery in Bond. On the sending side the Communicator constructs the message, converts it to external format and passes it to the Communication Engine. On the receiving side the Communicator gets the message from the Communication Engine, converts it to the internal format and delivers it to the destination.

```
                         true, cloader);
            return cl;
        }
        catch (ClassNotFoundException cnfe) { }
        catch(Exception ex) { }
        }
        return null;
    }
}
```

9.1.2 Communication Architecture

The communication architecture is presented in Figure 9.5 where we see the objects involved, the sender, the receiver, a pair of communicators that compose the message and a pair of communication engines that transport a message from one resident to another.

In this section we address the mechanics of message delivery and defer the problem of semantic understanding of messages for Section 9.1.3.1, when we introduce the concept of subprotocols.

Bond communication was designed with several objectives in mind:

(i) Support multiple external message formats for interoperability with other systems.

(ii) Hide the intricacies of message formatting and parsing.

(iii) Support multiple transport mechanisms for different levels of reliability and functionality. Delegate this function to a communication engine.

(iv) Support asynchronous communication and provide abstraction for delayed response. In a wide-area distributed system expect the response to a query or to a request for service to arrive only after a very long delay.
(v) Separate semantic understanding of messages from message delivery.The semantic understanding of messages should be done at the object level, various objects should have different levels of semantic sophistication. Stamp each message with an indicator allowing the recipient to determine with ease if it understands the message or not.

(vi) Support dynamic collections of semantically related objects.

(vii) Allow every object to receive messages. *Active objects* have a thread of control and can receive messages without any additional complications. *Passive objects* do not have a thread of control yet there are instances when it would be beneficial to receive messages. For example the model of an agent is an passive object containing the knowledge an agent has about the external world. Other agents should be able to send messages and update the model even when the agent is not running and the model is stored by a persistent storage server.

9.1.2.1 Message Delivery. The basic philosophy of the message delivery system is to transport a message for a remote object first to the resident hosting the object and then using the local directory to deliver it to the object itself.

In addition to the sending and receiving object two pairs of internal objects are involved: communicators and communication engines. The *communicators* are responsible to format the message and convert it from the internal to the desired external format on the sending side and perform a reverse transformation on the receiving side.

The *communication engines* transport a message. The communication engine on the sending side performs a multiplexing function, it may append additional information before delivering the message to its peer on the receiving side. The communication engine on the receiving side performs a demultiplexing function, it removes the additional information and then delivers the message to the communicator. The Distributed Awareness mechanism discussed in Section 9.1.2.10 relays on piggybacking control information on regular messages. The communicators are discussed in Section 9.1.2.6 and the communication engines in Section 9.1.2.7.

To construct a message for a remote object the communicator at the sending side needs:
(i) The contents and the dialect of the message (the subprotocol). They are provided by the sender object.

Table 9.2 Reserved names for the internal format of Bond messages.

Reserved name	Description
`sender`	`bondAddress` of message sender
`destination`	`bondAddress` of message destination
`reply-with`	Unique identifier, the destination object should use when replying to this message
`in-reply-to`	Unique identifier in the `reply-with` variable of a previous message, this message replies to.
`subprotocol`	The message is part of a subprotocol. If destination object does not understand the subprotocol, it answers `sorry`.
`performative`	The speech act of this message, question, answer, notification etc. as required by KQML specification.
`contents`	The contents of the message.
`piggyback`	Data field attached to the message to carry information between two directories or two communication engines.

(ii) The `bondAddres` of the resident and the `bondID` of the object. They are provided by the shadow of the remote object.

On the receiving side the message is delivered to the communication engine which uses the local directory to locate the destination object and then sends it to the communicator. Here the message is converted to the internal format and delivered.

9.1.2.2 Internal and External Message Format. The internal format of the messages is an unordered collection of *name-value* pairs implemented as dynamic properties of the `bondMessage` object . The *name* is a string, while the *value* can be any data type, including user-defined Java objects. There are four groups of *reserved names*, see Table 9.2, derived from the parameters of KQML messages.

-*Addressing variables:* the source, destination and optionally the retransmission objects location and id.
-*Message identifiers:* identify the message, request an answer (`reply-with`) or identify the question to which the current reply is a message (`in-reply-to`).
- *Semantic identifiers:* identify the context of the message. Bond favors the use of the `subprotocol` variable, but `language` and `ontology` might be used, especially in the case of interoperation with other systems.
- *Hidden variables:* variables attached to the message object during its lifetime but not delivered at the destination but removed either by the messaging thread or by preemptive probes, e.g. the `piggyback` variable, used by the distributed awareness mechanism [31].

Most of the parameters in Table 9.2 are added automatically to messages. The process of message annotation is summarized in Figure 9.5.

The internal format of Bond messages relies on the dynamic properties of Bond objects and cannot be used to communicate to other systems, thus we need an external representation for messages.

There are two external representations for Bond messages: KQML [19] and XML [47]. In both cases there is a one-to-one mapping between the internal and external format. The `XMLMessaging` variable determines the format of the messages delivered to the network. The parser method of the `bondMessage` object recognizes the format of a message and delivers it for parsing to either the internal KQML parser or to the external XML, xerces parser.

The system can use KQML and XML format messages simultaneously and it is possible to specify XML or KQML messages on a host-by-host basis thus objects may interact with KQML and XML-based systems simultaneously.

The conversion from internal format to a text-based external format implies a considerable performance penalty, somewhat higher for XML. For slow and medium speed networks, this penalty is hidden by the network latency, but for high speed networks the conversion overhead may have a significant negative performance impact. Whenever interoperability with other systems or readability of messages are not important, the system can be configured to send messages in a serialized version of the internal format.

The KQML composer transforms the internal format of Bond messages to valid KQML statements. The value of the `performative` variable is set as the performative of the KQML message. If there is no such variable, the `tell` performative is used. All other variables are set as parameters of the resulting message. If the type of the variable is not `String`, the variable is Java serialized into a byte buffer and encoded using the Base64 algorithm. Base64 encoded strings are prefixed with the "@@@" escape sequence, to allow the KQML parser to recognize them.

The KQML parser, `bondKQMLParser`, at the receiving side, parses a KQML message into `bondMessage` internal format, and decodes the embedded variables. Mapping name/value pairs to/from KQML is highly efficient. The KQML implementation in Bond is limited to syntactic parsing, the semantic interpretation is done by the object, using the internal format.

XML, the Extensible Markup Language [47] is a general-purpose information exchange format. An XML text consists of a *Document Type Definition (DTD)* followed by a series of potentially embedded *elements*. Each element is defined by a starting and an ending tag. A number of parameters can be specified, in the `name = value` format of a starting tag. This feature allows sets of name/value pairs to be mapped into XML format. `BondMessage.dtd` gives the rules to map Bond messages into XML:

```
<?xml encoding="US-ASCII"?>
 <!ELEMENT message> <!ATTLIST message
          performative CDATA #REQUIRED
          content CDATA #REQUIRED
          sender CDATA #REQUIRED
          destination CDATA #REQUIRED
          subprotocol CDATA #REQUIRED
          reply-with CDATA
          in-reply-to CDATA
>
```

The conversion into XML is done by the composer function, of the `bondMessage` object. XML messages are parsed by an external XML parser. Bond can use any XML parser conforming to the SAX event-oriented API. The performance of parsers varies, currently we use the *Apache-xerces* parser. A full featured XML parser is more complex then a KQML parser, thus parsing XML messages is less efficient then parsing KQML messages.

9.1.2.3 Synchronous Communication. Message-oriented distributed object systems support both asynchronous and synchronous communication. Systems based upon remote method invocation favor synchronous communication when the caller blocks until the call returns. Some systems based upon remote method invocation circumvent this limitation and allow remote method invocation without return values. They implement asynchronous method calls as a pair of synchronous method calls without return values.

Synchronous communication is supported in Bond by the `ask` and `waitReply()` methods. `ask()` is a synchronous version of `say()`, it automatically tags the message as one which needs a reply, and waits until the reply arrives, or a timeout occurs. `ask()` returns the reply, or `null` in case of a timeout as shown in the following example:

```
bondMessage question = new bondMessage(``(ask-one :
     content get :value i :)'', ``PropertyAccess'');
bondMessage rep = bs.ask(question, this, 10000);
if (rep == null)
    {
    System.err.println(``Timeout of 10s was exceeded'');
    } else {
    System.out.println(``Field i of remote object is''+
       (String)rep.getParameter(``value''));
    }
```

The `ask()` function blocks only the current thread of the Bond application, all other threads continue to run.

The `waitReply()` method is an alternative for synchronous communication allowing the execution of some code between message sending and the reply. The message should marked as needing a reply and sent using the `say()` method as in the following example:

```
bondMessage question = new bondMessage(``(ask-one
 :content get :value i :)'', ``PropertyAccess'');
question.needsReply();
bs.say(question, this);
...
```

```
code executed before the reply
...
rep = question.waitReply(10000);
```

9.1.2.4 Asynchronous Communication. Asynchronous communication is more difficult to implement then synchronous one, the system must be prepared to accept an incoming message at any time, regardless of its current state. Such an *active message* system is difficult to program, it must treat each message as an interrupt. In case of multiple messages it is difficult to pair the incoming message with the original request.

Bond provides a mechanism to pair incoming messages with the original requests. Messages which require an answer call the `needsReply()` function that creates a `reply-with` field in the message and attaches a unique identifier to it.

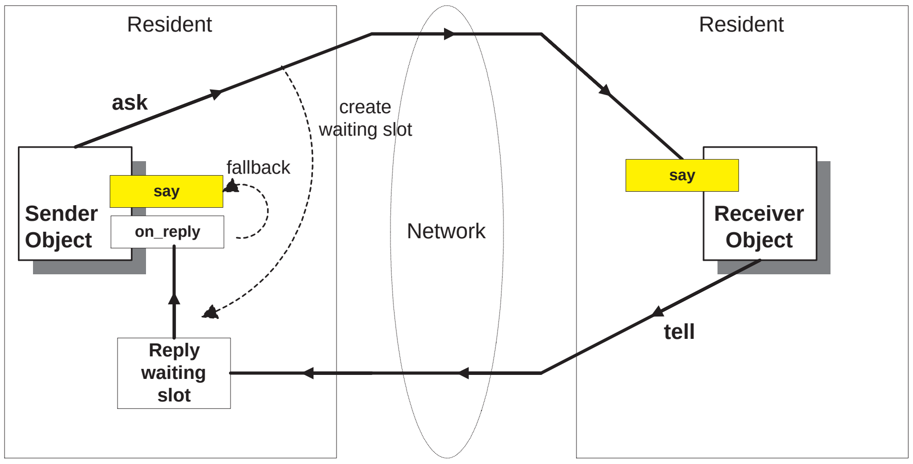


Fig. 9.6 Asynchronous communication. The sender object uses the `ask` performative with `reply-with` field to send an asynchronous message. The communicator creates a *message waiting slot* for the sender object and deposits there a copy of the original message and the unique id of the reply. Eventually the receiver object replays using the `tell` performative. When processing incoming messages, the communicator checks the waiting slot table for the unique id of the message and if a waiting slot exists, the communicator delivers the message to it. If the incoming messages has the `on_reply` field set then it is delivered directly, else it is delivered by the `say()` method.

Before sending a message with a `reply-with` field, the communicator creates a *message waiting slot* for the sender object. The message waiting slot contains the original message and the unique id of the reply. When processing incoming messages, the communicator checks the waiting slot table for the unique id of the message. If the message has a waiting slot, the communicator pairs the reply and question together and delivers them to the `on_reply` function:

```
public int on_reply(bondMessage message, bondMessage reply)
```

An object can isolate replies to earlier messages, from unexpected messages by catching the message in the `on_reply()` instead of the `say()` method. The `say()` method acts as a fallback for messages even if the object does not implement `on_reply` as shown in Figure 9.6.

9.1.2.5 The Subscribe-Notify Model for Event Handling. Java objects use the *listeners* abstraction to capture events. An object can *register* itself as a listener for a certain type of events. The object is notified every time the corresponding event occurs, until it decides to *un-register*. The object should implement a Java interface for the type of events it registered. The events are passed as procedure calls to the given interface.

Distributed object systems like CORBA extend this concept for objects that are not co-located. An *event* service allows an object to register itself as a listener for events generated by a remote object.

The *subscribe-notify* model used in Bond is an extension of the Java model for handling remote events in a distributed-object system. In this model an object expresses its interest in events associated with remote objects by subscribing to them and it is notified when the events occur. Throughout this presentation the object subscribing to an event is called a *monitor* while the object generating the event is called it monitored object.

In Bond events are generated when a property of a Bond object changes. Thus even passive objects may generate events. A property of an object stored by a Persistent Storage Server may be changed and the instance the change occurs, an event is generated.

Consider for example an application where an agent monitors the stock market and maintains several accounts. The portfolio managed by each account consists of many stocks. The owner of the account may request to be notified when the market value of the account goes below a threshold. In this example the agent queries periodically one of the servers providing market updates and modifies accordingly objects named `account` one for each customer. This object has a property called `Warning`, a boolean variable, with a default value `false`. This value is set to `true` when the condition requesting the user to be notified is met. If the owner of the account has subscribed to this property she will be notified immediately when this property changes. The `account` object may have multiple properties and the user could subscribe to any or to all of them.

When a Bond object decides to monitor a remote object it sends a message with the *subscribe* performative to that object. Internally, an *event waiting slots* is created automatically by the communicator of the resident hosting the monitor, as shown in Figure 9.7.

The object being monitored sends a message with the `tell` performative every time the corresponding property of the object changes. The monitor matches these messages against the set of event waiting slots. If a match is found, the message is delivered to the object by the `on_event` function. The event waiting slot is

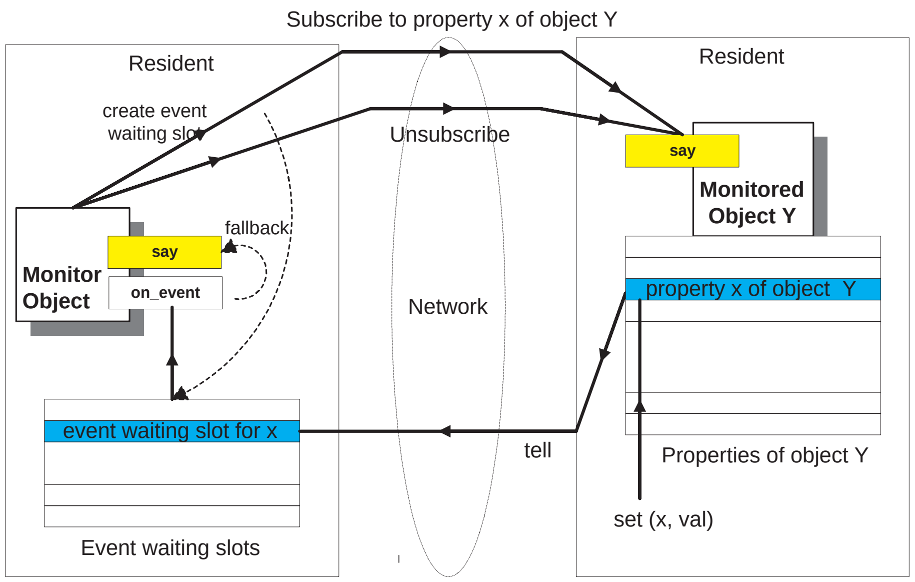


Fig. 9.7 The subscribe-notify event model. An object called a monitor may request to be notified when property `x` of object `Y` is modified. When the `subscribe` message is sent, the communicator of the monitor creates an event waiting slot.

automatically removed by the communicator object whenever the monitor sends an `unsubscribe` message.

A monitor can separate an event notification message from other types of messages. If the object does not implement the `on_event` function, the `say()` function is used as a fallback to deliver the message to the monitor.

The code for the `bondListener` follows:

```
public class bondListener extends bondObject
                          {publicbondListener() { }
 public void subscribeAsListener(String property,
                             bondObject listener) {
     Vector v;
     if (values == null)
        values = new Hashtable();
     try { v = (Vector)listeners.get(property); }
     catch (NullPointerException e) {v = new Vector();}
     v.addElement(listener);
     values.put(property, v); }
 public void unsubscribeListener(String property,
                             bondObject listener) {
     Vector v = (Vector)values.get(property);
     if (v == null)  return;
     else  v.removeElement(listener); }
 public void notifyListener(String property,
                                   Object value) {
      Vector v;
      if (values == null)  return;
      if ((v = (Vector)values.get(property)) == null)
         return;
      if (v.size() == 0) return;
      for (Enumeration e = v.elements();
                               e.hasMoreElements();) {
         bondListenerInterface bl =
         (bondListenerInterface)e.nextElement();
         bl.propertyChanged(property); } }
 }
```

The `set` function executed by the object being monitored notifies the monitor when a property subject to monitoring changes:

```
public synchronized Object set(String name, Object value) {
 Object ret = null;
  try {
    if (conf.useAccessors) {
        try {invokeSet(name, value);return value;}
        catch (InvocationTargetException ite2)
        catch (NoSuchMethodException nsme2) }
    try  {
        Field f= getClass().getField(name);
        f.set(this, value);
        ret = value; }
    catch (NoSuchFieldException nf)  {
        if (values==null) values=new Hashtable();
        if (value==null)  {
        values.remove(name);
        return null;    }
        values.put(name,value);
        ret = value;  }
  }
  catch (IllegalAccessException iae1)
  catch (IllegalArgumentException iae2)
  catch (NullPointerException npe)
  if (listeners != null) listeners.notifyListener(name, value);
  return ret;
 }
```

9.1.2.6 The Communicator. The function of the communicator on the sending site is to compose a message out of its components and to pass the message to the communication engine, see Figure 9.5. It fills in: the `sender` field with the `bondIPaddress` and the `bondID` of the sender; the `destination` field field with the `bondIPaddress` and the `bondID` of the destination; the `reply-with` field with a newly created identifier, if the message requires a reply; the waiting slots with the message or event identifier if waiting slots are needed. The pseudocode for the sending process of the communicator follows:

```
for(every sent message)
   if (external format)
      transform message in internal format
   endif
   annotate with the sender address
   if (reply needed)
      annotate with a unique reply-with field
      create a reply waiting slot
   endif
   if (performative is subscribe)
      create an event waiting slot
   endif
   if (performative is unsubscribe)
      delete the event waiting slot
   endif
   if (need to send info to destination)
      annotate with the piggyback field
   endif
   pass the message to communicator engine
```

On the receiving side the communicator extracts the components of a message delivered by the communication engine as shown in Figure 9.5. First the communicator converts the message to internal format, then checks if the message is expected: (i) searches the *message waiting slots* table to check if the message is a reply to an earlier message and if so the waiting slot is deleted and the message is delivered to the object paired with the original question; (ii) searches the *event waiting slots* table to determine an event notification.

Finally the communicator delivers the message. If the destination has a unique `bondID` the communicator searches the local directory and delivers the message to the object. If the destination is an alias, the communicator picks up at random one of the objects with the given alias and delivers the message to it. If the destination object can not be found, the communicator sends an error message.

The communicator uses a *thread pool* to deliver a message to an object. A thread pool is a collection of threads waiting to be activated. Whenever a message needs to be delivered, the communicator wakes up a thread, passes to the thread the message and a reference to the destination object as parameters, and calls the `say` function of the destination object in the newly activated thread. After the return of the `say` function, the thread goes back to the wait state.

This message delivery mechanism de-couples the communicator object from the processing of messages at the object level and allows multiple messages to be processed simultaneously. The default size of the thread pool is $nthreads = 10$. If more than $nthreads$ messages need to be processed at the same time, additional threads are created, they deliver the messages and then the thread pool return to its original size. The pseudocode for message delivery is:

```
for(every incoming message)
   parse message
   remove the piggyback field if any
   if (has in-reply-to field) and
      (in-reply-to field maches a reply waiting slot)
   then
      deliver to object waiting on the reply waiting slot
      delete the reply waiting slot
   else if (performative is tell) and
           (sender maches an event waiting slot)
   then
      deliver to object waiting on the event waiting slot
   else
      lookup the destination object
      if (destination is alias)
         select an object with the alias at random
      else
         look up the object in local dir
      endif
      if (no object)
         send error message to sender
      wake up a thread in the threadpool
      deliver the message using the thread
   endif
end for
```

9.1.2.7 Communication Engines. A communication engine transports messages from one resident to another. The engine runs at a known port at a host with a given `bondIPaddress`. Currently the system comes with four interchangeable communication engines:

UDP communication engine based upon the UDP protocol. Datagrams do not require connection establishment or acknowledgements thus the UDP engine is faster then the TCP engine supporting a reliable connection-oriented protocol. Message size is limited to 64KB and there is no guaranteed delivery.

TCP communication engine based upon the TCP protocol. Its advantage is the unlimited message size and the guaranteed delivery.

Infospheres communication engine based on the `info.net` package from the Infospheres system [14]. Message size is limited to 32KB.

Multicast communication engine based upon the IP multicast protocol. It is used when the same message must be sent to a number of objects in a virtual object network.

Currently the system does not support the concurrent use of multiple communication engines.

Each engine has two methods to send, one for messages and another for objects and one method to receive messages. Before sending an object with the `realize()`

function the object is converted to a string and then it is encoded. The skeleton of the code to send messages and objects follows:

```
public void send(bondShadow bs,bondMessage m) {
  String mes = m.compose();
  try{
    InetAddress targetIP = InetAddress.getByName
                   (bs.remote_address.ipaddress);
    myUDPDaemon.send(targetIP, bs.remote_address.port, mes);}
    catch(UnknownHostException e){e.printStackTrace();}
  }
public void sendObject(bondShadow bs, bondObject bo,
                                String in_reply_to) {
  bondExternalMessage bm = new bondExternalMessage();
  bm.in_reply_to = in_reply_to;
  bm.bo = bo;
  String m = Base64.Object2String(bm);
  try{
    InetAddress targetIP = InetAddress.getByName
                   (bs.remote_address.ipaddress);
    bondUDPDaemon.send(targetIP, bs.remote_address.port, m);}
    catch(UnknownHostException e){e.printStackTrace();}
  }
```

Each communication engine has one daemon responsible to send a receive a message using the transport protocol for that engine. The skeleton of the `bondUDPDaemon` follows:

```
public class bondUDPDaemon extends bondObject {
  public bondUDPDaemon(int port) throws SocketException {
    super(false);
    udpSocket = new DatagramSocket(port);
    localport = port; }
  public int getLocalPort(){return localport;}
  public void send(InetAddress targetIP,
                int targetPort, String m) {
     bufOut = m.getBytes();
     udpOutPacket = new DatagramPacket(bufOut,
          bufOut.length, targetIP, targetPort);
     try{udpSocket.send(udpOutPacket);}
     catch(IOException e){}
  }
  public String receive() {
    try{
      udpInPacket = new DatagramPacket(bufIn, 65535);
      udpSocket.receive(udpInPacket);
      InetAddress fromAddress = udpInPacket.getAddress();
      fromHostname = fromAddress.getHostName();
      fromPort = udpInPacket.getPort();
      String mes = new String(udpInPacket.getData(),
                              0, udpInPacket.getLength());
      return mes;}
    catch(IOException e){ return null; }
  }
  public String getFromHostname(){
    return fromHostname; }
  public int getFromPort(){
    return fromPort;}
}
```

9.1.2.8 *Virtual Networks of Objects.* Distributed systems frequently contain groups of objects semantically related to one another such as: local directories of various residents; the groups of objects monitored by a single monitor; the group of sensors connected to a single data collector. These groups may overlap, an object may be a member of multiple groups. The members of a groups may receive multicast messages, and may be created and destroyed together even though they may be distributed across several residents.

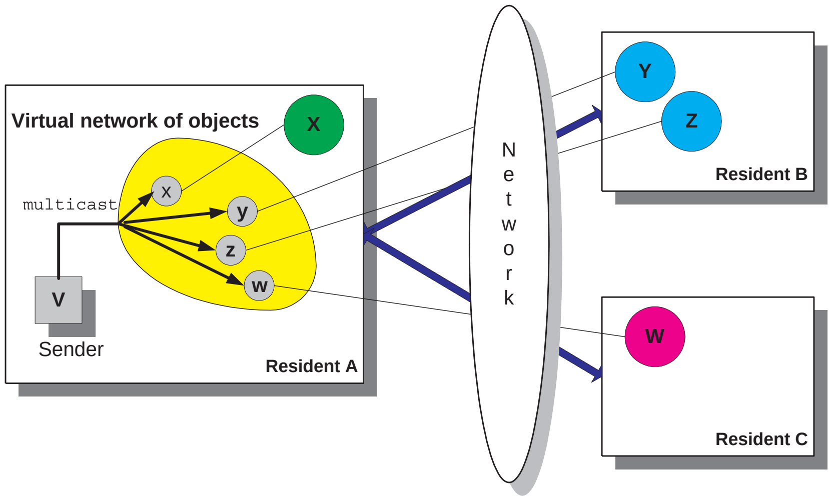


Fig. 9.8 An object `V` multicasts to a virtual network of objects. The virtual network of objects consists of the shadows `x,y,z,w` of objects `X,Y,Z,W`. Each shadow has the `bondAddress` and the `bondID` of the object. Thick lines connecting the residents indicate transport paths through the network.

In Bond we have an abstraction called a *virtual network of objects* for a group of semantically related objects. A virtual network of objects consists of the shadows of the objects as shown in Figure 9.8. The `bondVirtualNetwork` object supports primitives for:

(i) Objects to join and leave a virtual network.

(ii) Testing if the objects in a virtual network are alive, it automatically partitions the objects into two groups, *live* and *dead*.

(iii) Multicasting to the objects of the virtual network. If the application has the multicast communication engine installed, the message is transmitted using IP multicast. If the multicast engine is not available, or the location of the objects does not allow IP multicast, the multicast results into a sequence of unicasts.

The system could use virtual networks to connect local directories of residents to a global directory.

9.1.2.9 Object Mobility. The system provides a simple way of moving objects across the network, using the `realize` function applied on shadows. The sequence of operations needed to bring a remote object to the current resident is: (i) create a shadow of the remote object (either by knowing its name and location or by using the directory service) and (ii) call `realize()` method on the shadow.

The object mobility using the `realize()` function is triggered on the receiving side (*pull mode*), and does not require a cooperating entity on the sending side, see Figure 9.9.

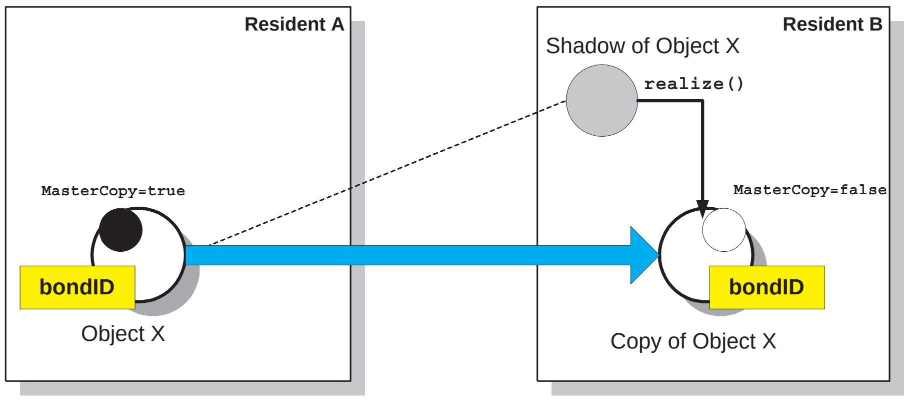


Fig. 9.9 Object mobility. The `realize()` function supports the creation of a local copy of a remote object. The original object and the copy have the same `bondID`, but the original has the `MasterCopy` boolean property set to `true` while the copy has it set to `false` .

A problems raised by the mobility is the *consistency of the copies*. The `realize()` function creates a remote copy of the object, with the same `bondID` as the original, and tags the moved object by setting its `MasterCopy` boolean variable to `false`. If the new object is modified, then two different copies of the same object exist. There are several ways of handling this problem:

(i) Physically move the object, discard the original object immediately after the move, and make the new object the master copy.

(ii) Clone the object, assign a new `bondID` to the copy immediately after the move.

(iii) Synchronize copies of the object to the master copy.

9.1.2.10 Distributed Awareness

Distributed awareness is a passive mechanism for the nodes of a message–passing distributed system to learn about the existence of other nodes without the need to communicate explicitly with them through *gossiping*.

In Bond each resident maintains an *awareness table* and exchanges the information in this table with other residents at the time of regular message exchanges between objects. This mechanism can be turned off at the start up time.

An entry in the awareness table contains:

(i) `bondAddress` of a resident,

(ii) `lastHeardFrom`, the time when we last heard from the resident, and

(iii) `lastSync` the time when the awareness information was last sent to the resident.

The awareness information is piggybacked onto regular messages exchanged between two residents as shown in Figure 9.5.

9.1.3 Understanding Messages

How do humans understand each other? Try to ask a total stranger the question "how many 5-fold axes does an icosahedron have" in Swahili. After a few trials you will realize that in order to communicate with one another, two individuals have to find some common ground, first they have to speak the same language, then they have to have some common domain knowledge.

How do objects in a distributed system understand each other? A solution is to have some public service where each object deposits a note describing the methods it can perform. CORBA uses such a service called "interface repository".

Table 9.3 Subprotocols.

Subprotocol	Function
Property access	Read/write access to properties of a Bond object.
Security	Establish trust relationship amongst Bond objects.
Monitoring	Monitor an object.
Agent control	Start, stop, and control a remote agent.
Scheduling	Schedules a contract
Persistent Storage	Save/load objects to/from Persistent Storage
Data Staging	Move files
Registration	Register a resident with the `SystemMonitor` and the `Directory Server`

An open system consists of a continuum of objects ranging from simple objects like an icon to complex ones such as a server or an agent. Moreover some objects are created dynamically or may acquire new properties dynamically. The sender of a message expects the receiver to understand and then to react to the message. This expectation places a rather heavy burden upon the objects of an open system. In closed systems the semantic gap can be closed, objects may agree to communicate only after some prior agreement as in the case of CORBA.

In Bond we partition the set of messages into "dialects" called subprotocols; two objects may communicate with one another if and only if they implement a common subset of subprotocols. Before delivering a message the `say()` method examines the subprotocol field of the message and it only delivers the message if the destination object, one of its ancestors, or a probe attached to the object implements the subprotocol.

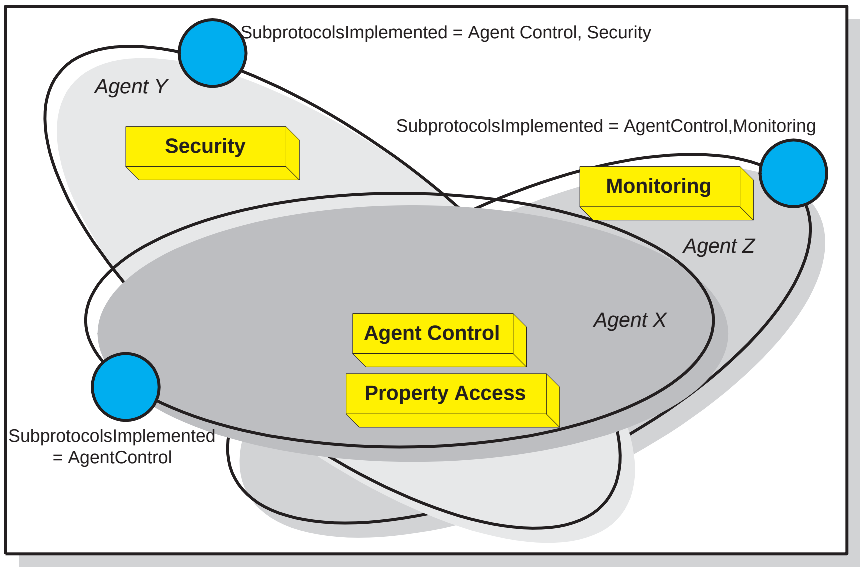


Fig. 9.10 Each Bond object has a property called `SubprotocolsImplementes` that lists the subprotocols implemented by the object. All Bond objects implement the Property Access subprotocol. All agents including `X, Y, Z` implement the *Agent Control* subprotocol. In addition agent `Y` implements the *Security* subprotocol, and agent `Z` the *Monitoring* subprotocol.

In this section we first introduce the concept of a subprotocol, then introduce static subprotocols and subprotocol inheritance in Section 9.1.3.2 followed by a discussion of dynamic subprotocols and probes in Section 9.1.3.3. We examine message delivery and the property access subprotocol implemented by all Bond objects in Sections

9.1.3.4 and 9.1.3.5. We conclude with a presentation of the configuration mechanism in Section 9.1.3.6.

9.1.3.1 Subprotocols. The set of Bond messages is partitioned into small, closed subsets of commands necessary to perform a specific task, called *subprotocols*. Each message identifies the subprotocol the message belong to, thus an object can decide if it understands the message or not.

Closed means that commands within a subprotocol do not reference commands outside it. The reply is always a member of the same subprotocol with the question. The only exception to these rules are the `(sorry)` and `(error)` performatives, valid replies to messages of any subprotocol.

Every Bond object implements at least the *property access subprotocol* which allows to interrogate and set the properties of another object. A typical object implements a number of subprotocols. Table 9.3 lists a subset of Bond generic subprotocols.

If two objects do not have any knowledge about each other, they interrogate the `SubprotocolsImplemented` property of each other and learn what subprotocols each of them implements. Then they can communicate using the intersection of the two sets.

Some subprotocols are *static*, they are available at the time an object is created, others are *dynamic*, added to an object as needed during the lifetime of the object. Subprotocols can also be created automatically as discussed later in Section 9.1.3.3.

9.1.3.2 Static Subprotocols and Inheritance. A Bond object inherits the subprotocols implemented by the objects above it in the object hierarchy. The message thread of a resident delivers an incoming message to the `say()` function of the object. If the message is not understood by the `say()` function of the object, it is then passed to the `say()` function of the immediate ancestor in the object hierarchy and this process continues recursively until either an ancestor that implements the subprotocol of the message is found, or the `say()` function of the `bondObject`, the root of the hierarchy answers `sorry`.

Figure 9.11 shows two examples of messages delivered to a `bondScheduler` object. This object extends a `bondAgent`, which in turn extends a `bondExecutable`, which in turn extends a `bondObject`.

The scheduler agent understands an agent control message because it inherits the agent control subprotocol from the `bondAgent`. The agent control message is delivered by the `bondAgent.say()` function. On the other hand, the scheduler agent is unable to understand a monitoring message, neither `bondScheduler.say`, `bondAgent.say`, `bondExecutable.say`, nor `bondObject.say` can deliver this message thus the reply is `sorry`. To understand a monitoring message an object must inherit the *monitoring* subprotocol from a `bondScheduler` object.

9.1.3.3 Dynamic Subprotocols and Probes. Some members of a class of objects may have functions and requirements different than those of the majority of objects in that class. For example, an agent may need to monitor other objects, or may have very strict security requirements. Yet, requiring all agents to understand

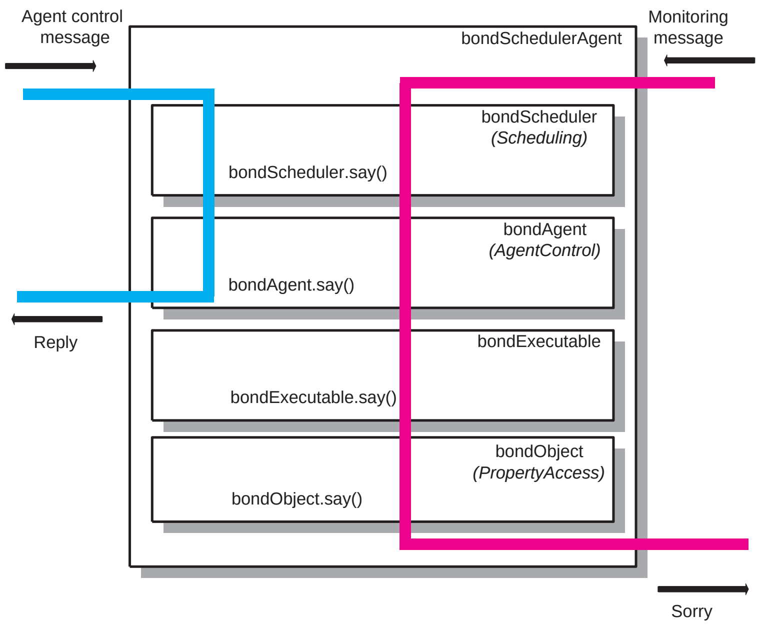


Fig. 9.11 The `bondSchedulerAgent` inherits the subprotocols of his ancestors, `bondScheduler, bondAgent, bondExecutable, bondObject`. The subprotocols implemented by each ancestor are in parenthesis. An agent control message is delivered by the `bondAgent.say()` function. On the other hand, the scheduler agent is unable to understand a monitoring message, neither `bondScheduler.say`, `bondAgent.say`, `bondExecutable.say`, nor `bondObject.say` can deliver this message thus the reply is `sorry`.

the monitoring and the security subprotocols imposes an unnecessary overhead for those who do not need to monitor or do not need additional security.

In Bond we have specialized objects called *probes* that are attached to a regular Bond object as a dynamic property. The only function of a probe is to understand a subprotocol. A Bond object implements all static protocols on its sub-tree of the Bond object hierarchy and all sub-protocols supported by the probes attached to it after the object was created.

This construction is similar in scope to the Decorator design pattern [18], it extends dynamically the functionality of an object without sub-classing. However the implementation is different, instead of a wrapper which captures the function call, we append dynamically an object.

Another object-oriented structure which allows objects to acquire new functionality after "programming time" is the notion of a *mixin* [11]. Mixins are generally implemented as abtract classes, with reserved functions for future functionality. As such, the programmer needs at least a rough idea about the nature of the functionality

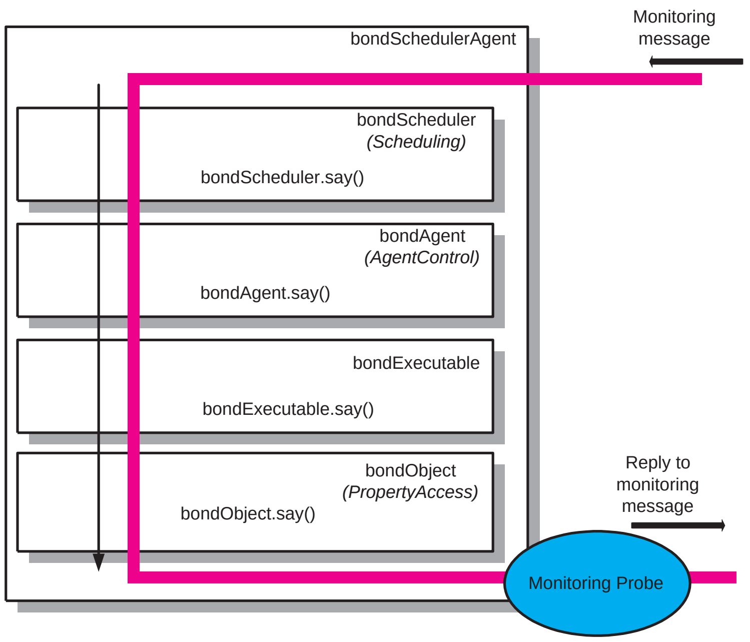


Fig. 9.12 A `bondScheduler` object extended with a monitoring probe. Now the object understands the monitoring sub-protocol and gives a meaningful reply to a monitoring message.

with which the object may be extended. In our case, the probes offer greater flexibility and adds the cost of the time to interpret syntactically and semantically the message.

The implementation of the `bondObject` guarantees that when an object does not understand a message, its dynamic properties list is searched for a probe which can handle the subprotocol and then deliver the message to the object. If no probe is found, the object replies `sorry`.

Two commonly used probes are the `bondMonitoringProbe` which understands the monitoring subprotocol and the `bondSecurityProbe` which allows an object to understand encrypted messages.

Figure 9.12 shows the same scheduler agent, this time extended with a monitoring probe. The probe implements the monitoring subprotocol. An incoming message in the monitoring subprotocol is passed down the inheritance hierarchy without being delivered to the object. At the `bondObject` level, we first check that the message does not belong to the property access subprotocol. Then we check the list of dynamic properties and find a probe that understands the monitoring subprotocol. The message is delivered to the probe that produces a meaningful reply.

In our system there are three types of probes:

(i) *regular* - activated after searching the list of the static subprotocols understood by an object, e.g., the monitoring probe,

(ii) *preemptive* - activated before searching the list, e.g. the security probe, and

(iii) *autoprobes* - used to load dynamically a probe at runtime.

The skeleton of the code for the `bondAutoProbe` is listed below. The `say` function parses the message and identifies the subprotocol, then examines a hashtable of probes and if one is found, the probe is loaded and the message is delivered to it.

```
 public class bondAutoProbe extends bondProbe {
  Hashtable lookup;
  public bondAutoProbe(bondObject parent) {
    super(parent);
    lookup = new Hashtable();
    initDefaults();
  }
 public void initDefaults() {
    addAutoLoad("Monitoring","bondMonitoringProbe");
    addAutoLoad("AgentControl","bondAgentFactory");
  }
 public void addAutoLoad(String name, String probename){
    lookup.put(name, probename);
  }
 public boolean implementsSubprotocol(String name) {
   if (lookup.get(name) != null) { return true; }
     return false;
  }
 public void say(bondMessage m, bondObject sender){
     String name = (String)m.getParameter(":subprotocol");
     String val = (String)lookup.get(name);
     bondProbe p = loader.loadProbe(val);
     p.parent = parent;
     parent.set("AutoProbe_"+name, p);
     p.say(m,sender);
  }
}
```

9.1.3.4 Message Sending and Delivery. Any Bond object can send and receive messages using the `say()` method. The `say` function, defined at root of the `bondObject` hierarchy, has the following signature:

```
  public void say(bondMessage m, bondObject sender) {
    if (sender == null) {
      sender = m.getSender();
    }
    String sp = m.getSubprotocol();
    if( sp != null ){
      if (sp.equals("PropertyAccess")) {
        sphPropertyAccess(m,sender);
        return; }
    }
    else {
      switch (m.performative) {
      case bondMessage.PF_SORRY:
      case bondMessage.PF_ERROR:
      case bondMessage.PF_DENY:
    return;
      default:
      }
    }
    if (values != null) {
      bondAutoProbe ap = null;
      for (Enumeration e = values.elements();
                                e.hasMoreElements();) {
    bondObject o = (bondObject)e.nextElement();
    if (bondProbe.class.isAssignableFrom(o.getClass())
        && o.implementsSubprotocol(sp)) {
      if (o instanceof bondAutoProbe) {
        ap = (bondAutoProbe)o;
      } else {
        o.say(m,sender);
        return; }
    }
    if (ap != null) { ap.say(m,sender);}
    }
  }
```

The `say` method is overwritten to support specific features for individual classes of objects. The message processing ability of an object is inherited in the object-oriented sense. At the end of the overwritten `say` method of an object there is a fallback to the `say` function of its immediate ancestor.

9.1.3.5 The Property Access Subprotocol. The *property access* is the subprotocol implemented by every Bond object. This subprotocol is used to read and write properties of another object. Table 9.4 lists the messages in this subprotocol.

The *performative* gives the broad meaning of the message. For example `ask-one` is a question requesting an answer, `achive` is an imperative request, `tell` is the response to a question. The *content* specifies the actual function requested, for example `set,get` are used to store and respectively read a property. The parameters provide command specific information.

When we set the value of a property, then the *new_value* is either a string, or a BASE64 encoded value. In a reply to `get` the *value* is either a string or a BASE64 encoded value; if there is no such property, *value* is a BASE64 encoded `null`. A reply to `set` confirms setting the property and it is sent only if `needsReply()` was invoked on the `set` message.

If object `X` wants to obtain the value of the property `x` of object `Y`, it sends the following message:

```
(ask-one :sender X :receiver Y :subprotocol PropertyAccess
```

Table 9.4 The messages of the property access subprotocol. A subprotocol is a closed set of messages. A message consists of a *performative* indicating the broad meaning of the message, a *content*, and parameters.

Performative	:content	Parameters	Description
ask-one	get	:property *name*	Get value of property *name* of remote object.
achieve	set	:property *name* :value *new_value*	Set value of property *name* of remote object to value *new_value*.
tell	value	:value *value*	Reply to `get`. *value* is the value of the requested property.
tell	ok		Reply to `set`. Confirms setting the property.
sorry		:error *error-name* :description *description*	An error occurred.

```
          :content get :property x :reply-with zzzz)
```

Assuming that property `x` of object `Y` has value `7`, then object `Y` replies with the following message:

```
(ask-one :sender Y :receiver X :subprotocol PropertyAccess
          :content value :value 7 :in-reply-to zzzz)
```

The property access subprotocol supports the `set`, `get` and `realize` functions.

```
void sphPropertyAccess(bondMessage m, bondObject sender){
   switch(m.performative) {
   case bondMessage.PF_ASK_ONE:
     if (m.content.equals("get")) {
       Object val = get((String)m.getParameter(":property"));
       bondMessage rep = m.createReply("(tell
           :subprotocol PropertyAccess :content value)");
       rep.setParameter(":value", val);
       sender.say(rep,this);
       return;}
     if (m.content.equals("set")) {
       set((String)m.getParameter(":property"),
                         m.getParameter(":value"));
     if (m.expectsReply()) {
       m.sendReply("(tell :content ok)", this);}
       return;}
   case bondMessage.PF_TELL:
```

```
# The system expects to find this information in:
# Bond/bond/core/properties
bond.debug=false
bond.agentLazyLoading=true
bond.strategy.repository=http://olt.cs.purdue.edu:8001/Bond/
bond.distributedAwareness=false
bond.communicationengine=UDP
bond.UDP.port=2000
bond.TCP.port=2000
bond.scheduler=RR
bond.microserver.enable=false
bond.microserver.port=2099
bond.filelogger = yes
default.monitoring.agent=Agent1+danube.cs.purdue.edu:2000
bond.faultDetection=true
```

Fig. 9.13 A sample properties file.

```
      if (m.content.equals("realize")) {
         com.sendObject((bondShadow)m.getSender(),
         this, (String)m.getParameter(":reply-with"));
         return;}
      if (m.content.equals("ok")) { return;}
      return;
      return;
    }
  }
```

9.1.3.6 Bond Configuration. At start-up time the system reads a file describing the desired system properties and creates a configuration object, `bondConfiguration`. A sample `properties` file is shown in Figure 9.13.

Most of the system properties in this file are self-explanatory. Here we only note that agent strategies can be loaded when an agent is activated, or can be deferred until the strategy is actually needed, an option controlled by the setting of the `bond.agentLazyLoading` variable. A `bondStategy` is a procedure activated when an agent enters a state as described in Section 9.2. A strategy repository is a database of common strategies. Distributed awareness, see Section 9.1.2.10 is a feature allowing residents to learn about each other. Bond agents support several schedulers for their actions, one of them being Round Robin, RR.

A microserver is an object that understands the HTTP protocol and is capable to access the properties of an object via a Web browser. All events can be logged on a file if so desired and a resident may request to be monitored by a running agent. The fault detection features may be activated at the start-up time.

The `bondConfiguration` object creates a running environment tailored to the options in the `properties` file.

9.1.4 Security

9.1.4.1 Introduction Security is an important concern for any network environment, as information in transit is vulnerable, and the use of resources in different administrative domain introduces issues of trust and consistency between them. A distributed object system poses new challenges to security mechanisms. For example security auditing should be able to identify correctly the principal, the original sender of a request, even after a chain of calls involving multiple objects. There is also the need of delegation, the propagation of attributes of the principals between components. Delegation allows one component to act on behalf of a principal.

Applications of network computing have vastly different security requirements and the trade-off between security and performance is application specific. It is infeasible to consider one security model suitable for all applications and all environments. Additional security challenges posed by network computing are discussed below. The user population and the resource pool are large and dynamic. A user may only be aware of a small fraction of the components involved in a computation.

The relations among components may be rather complex, a component may act both as a server and a client at the same time. Traditional distributed systems use RPC or TCP/IP as their primary communication mechanism. In contrast, a distributed computing environment may use two-sided communication mechanism like message passing, streaming protocols, multicast, and/or single-sided get/put operations, as well as RPC. Components may communicate through a variety of mechanisms.

The boundaries of trust are more intricate because of dynamic characteristic of components. The trust users have in components is threatened when components can be mobile between hosts and new components can be created on the fly. Boundaries of trust are more complex because an activity typically involves multiple domains with different security policies and security models. Computation may be distributed to many more machines than any given user has control over.

Granularity, consistency, scalability, flexibility, heterogeneity and performance are important aspects of distributed object security. A security design implies trade-off among these requirements. For example strong security and good performance are competing requirements. Coarse-grain security is easier to manage than fine-grain.

In Bond we opted for an extensible core object that can support multiple security models and can be added dynamically to existing object. This philosophy leads to several design principles:

(i) Provide a framework for security, not force an implementation. Bond leaves the decision of choosing the format of credentials, the authentication policy, the access control policy, and so on, to the system developer or the system administrator. Bond security is implemented as an extensible core Bond object called `BondSecurityContext` and a set of well-defined security interfaces.

(ii) Separation of concerns, various aspects of a complex object design, including security, should be separated from one another. In the initial design and implementation phase the creator of an object should only be concerned with functionality. Once the object is fully functional the creator needs to investigate the security requirements and augment the object with the proper security context by including a *probe* called BondSecurityContext. This dynamic property of a Bond object sets up a secure perimeter for the object, it intercepts all incoming and outgoing messages and enforces the security and access control models selected by the creator of the object.

(iii) Support multiple authentication and access control models. This goal is achieved by defining a common interface for different security functions, like credential, authentication and access control.

The Bond security framework is based upon the concept of *preemptive probe* discussed in Section 9.1.3.3. The preemptive probe is activated before any attempts are made to deliver the message to the object, it intercepts all messages sent to the object.

9.1.4.2 Security Models. Security in a network environment includes authentication and access control. *Authentication* refers to the process of identifying an individual. *Access Control* is the process of granting or denying access to a network based on a two-step process, authentication to ensures that a user is who he/she claims to be, and access control policy which allows the user access to various resources based on the user's identity.

Some of the authentication models are:

PAP - Password Authentication Protocol. The most basic form of authentication, the user's name and password are transmitted over the network and compared to a table of name-password pairs. Typically, the stored passwords are encrypted.

CHAP - Challenge Handshake Authentication Protocol. The authentication agent, typically a network server, sends the client program a key to encrypt the username and the password.

Kerberos - ticket-based authentication. The authentication server assigns a unique key, called a ticket, to each user that logs on to the network. The ticket is then embedded in every message to identify the sender of the message.

Certificate-based authentication. This model is based on public key cryptography. Each user holds two different keys: public and private. The user can get a certificate that proves the binding between the user and its public key from a third party. The private key is used to generate evidence that can be sent with the certificate to server side. The server uses the certificate and evidence to verify the identity of the user.

Credential is a secret code that proves the identity of an individual. Authentication models use different credentials, e.g. username/password in PAP and CHAP, user identifier/ticket in ticket-based authentication, and user certificate/private key in the certificate-based authentication.

Access control models include firewall and access control list, ACL. Firewall grants or denies access based upon the IP address of the requester. An access control list specifies what operations a user may perform on each resource.

Table 9.5 Authentication models

Type	Interface	Authenticator Interface
Name & Pass	`bondPAPCredentail`	`bondPasswordAuthenticator`
CHAP	`bondCHAPCredential`	`bondChallengeAuthenticator`

9.1.4.3 Implementation. `BondSecurityContext` is a preemptive probe that establishes a defense perimeter for the object it is attached to, by intercepting incoming and outgoing messages with two methods: `incomingMessageProcess()` and `outgoingMessageProcess()`. In addition there are several security interfaces:

`BondCredentialInterface` - defines the method to access the credential possessed by the current `BondSecurityContext`. This interface provides two groups of methods: (i) Methods to respond to authentication request from a remote object. Usually a challenge is contained in the authentication request, and the response is derived from both the challenge and the information provided by the credential. The response is generated differently depending on security models. (ii) Methods to generate a user identifier and a proof to be embedded in each outgoing message and prove to the receiver the identity of sender. The proof has different meaning in different security models. In a username/password model, the proof can be a password, or an encrypted password, in a ticket based security model, the ticket itself can be a proof, in a certificate–based model, the evidence generated by encrypting a random string with the private key can be an eligible proof.

`BondAuthenticatorInterface` - defines the authentication method for each message received by an object. The developer or the administrator may deploy one of the authentication models mentioned earlier. The only restriction is to adhere to this interface. The `authenticateClient()` is the only method provided by this security interface. This method returns an authenticated user identifier. This identifier can be used for access control or auditing.

`BondAccessControlInterface` - defines the access control method for each message received by an object. The methods provided by this security interface are `initACL()` and `checkRight()` based upon the authentication models discussed earlier.

The code below illustrates the implementation of `bondSecurityContext` that supports authentication and access control in the `incomingMessageProcess()`.

```
bondSecurityContext extends bondProbe {
 private bondCredentials bcs;
 private bondAuthenticatorInterface bau;
 private bondAccessControlInterface bac;
/* incomingMessageProcess is called by the message
```

Table 9.6 Access control models

Type	Access Control Interface	Required Authenticator
IP	`bondIPAddressAccessControl`	-
ACL	`bondNameBasedAccessControl` `bondRightBasedAccessControl`	`bondChallengeAuth` `bondPasswordAuth`

```
  thread on each received message */
 public void incomingMessageProcess(m, sender){
/*1.authenticate message */
  authenticated_user_id
   = bau.authenticateClient(m);
  if( authenticated_user_id == null ){
    sender.say( sorry message );
    return;}
/* 2.enforce access control */
  result = bac.checkRight(authenticated_user_id,m);
  if( result == false){
    sender.say( sorry message ); return;}
/* 3.pass the message to the object */
  parent.say(m, null); }
```

The code also shows several objects that implemented the security interfaces defined above.

Table 9.5 lists the authentication models and Table 9.6 lists the access control models implemented in Bond.

All authenticators in Table 9.5 need an Authentication Server maintaining the usernames and the passwords. If the service provider uses one type of authenticator, the client should use the corresponding credential to make the authentication successful.

9.1.4.4 Examples. The following examples illustrates how to construct secure objects. Assume that we have one client, two generic servers and an authentication server that provides account management and authentication services.

The client uses an existing account (`uid=hector` and `passwd= ham`) to access services provided by the two servers. One of them, *serverA* enforces plain password-based authentication and firewall-based access control, while *serverB* enforces CHAP-based authentication and name–based access control.

The code below shows how to setup *serverA* as a secure object enforcing plain password-based authentication and firewall-based access control.

```
/* create a new servr object, serverA server */
  serverA = new server();
/* create a plain-password based authenticator */
 bondPasswordAuthenticator bau  = new
```

```
          bondPasswordAuthenticator(baserver);

/* create a firewall-based access controller AC */
 bondIPAddressAccessControl bac = new
                      bondIPAddressAccessControl();
 bac.initACL("firewall.acl");
/* create a security context */
 bondSecurityContext gatekeeper = new
                     bondSecurityContext(serverA);
/* set the AC and authenticator of context */
  gatekeeper.setAccessControl(bac);
  gatekeeper.setAuthenticator(bau);
/* set the security context into serverA */
  serverA.setSecurityContext(gatekeeper);
```

The format of the access control list `firewall.acl` is:

```
* Firewall configuration file consisting
* of pairs <hostname><mask>
dragomirna.cs.purdue.edu 255.255.255.0
```

Hosts in the same sub-net with the machine `dragomirna.cs.purdue.edu` can access *serverA*.

The code below shows how to create a secure object enforcing CHAP-based authentication and name-based access control, AC.

```
/* create a new server object, serverB */
 server serverB= new server();
/* create a CHAP-based authenticator */
 bondChallengeAuthenticator bpau  = new
           bondChallengeAuthenticator(baserver);
/* create name-based AC and initialize it */
 bondNameBasedAccessControl bac = new
                bondNameBasedAccessControl();
 bac.initACL("names.acl");
/* create a security context for serverB */
 bondSecurityContext gatekeeper = new
                bondSecurityContext(serverB);
/* set the access controller and authenticator */
 gatekeeper.setAccessControl(bac);
 gatekeeper.setAuthenticator(bpau);
/* set security context as dynamic property of serverB*/
 serverB.setSecurityContext(gatekeeper);
```

The format of the access control list file `names.acl` is:

```
*
* Name based ACL, the format of
* this file is as following
```

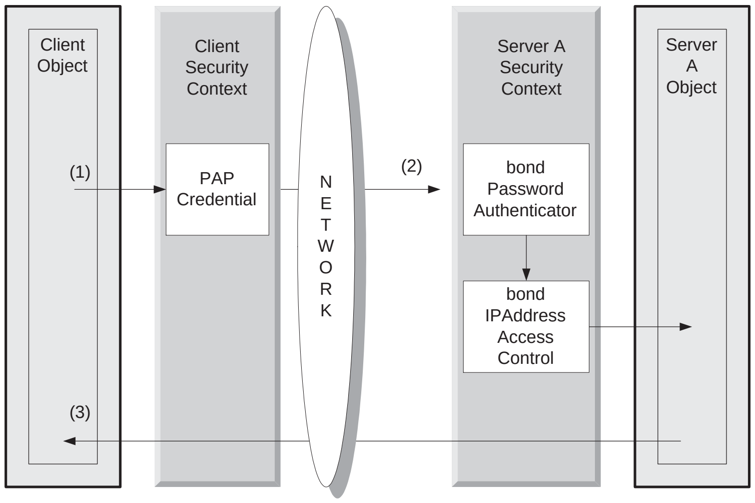

Client Object
Client Security Context
PAP Credential
NETWORK
Server A Security Context
bond Password Authenticator
bond IPAddress Access Control
Server A Object
(1)
(2)
(3)

Fig. 9.14 Processing of a service request using a PAP model. (1) is the original service request from the client. (2) is the service request with a username and password added by the client's security context. (3) is the response to the service request from the client if the security context of the server validates the credentials.

```
* <name> <right1,right2,rightN>
hector persistent-object-read,persistent-object-write
```

This means that user `hector` is allowed to save objects to and reload them from this server.

The parameter, `baserver`, is used to create the authenticators in both cases. This means serverA and serverB share the account information stored by the `baserver`, the authentication server of the domain.

To set up a client as a secure object:

```
/* create a client object */
 client clio = new client();
/* create a security context for client */
 bondSecurityContext bsc = new bondSecurityContext(clio);
/* setup a PAP credential */
 bondPAPCredential bc1 = new bondPAPCredential
      ("hector","ham"); bsc.setCredential(bc1,"serverA");
/* setup a CHAP credential */
 bondCHAPCredential bc2 = new bondPAPCredential
      ("hector","ham"); bsc.setCredential(bc2,"serverB");
/* setup this security context for client */
 clio.setSecurityContext(bsc);
```

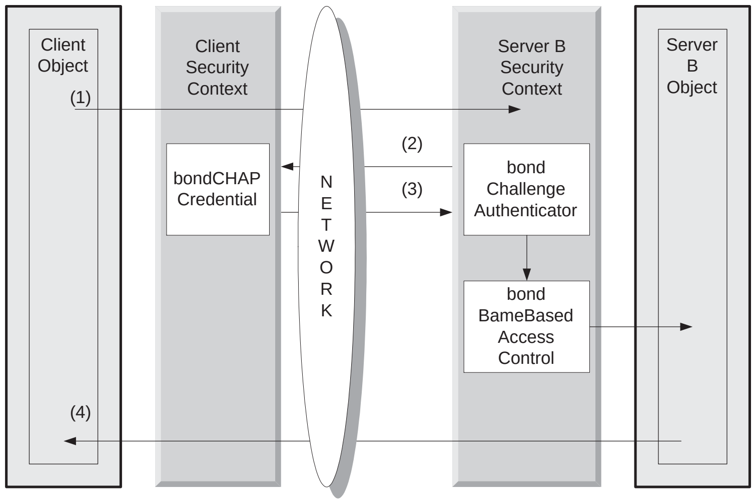


Fig. 9.15 Processing of a service request using the CHAP credential. (1) is the original service request from the client. (2) is a challenge generated by the security context of the server. (3) is the response to the challenge. (4) is the response to the service request from the client if the security context of the server validates the credentials.

Once properly set up, `bsc` adds appropriate credentials to outgoing requests by checking destinations. In the above example, requests to serverA are associated with bondPAPCredential, while those to serverB are with bondCHAPCredential.

A scenario involving the interaction between the client and *serverA* is shown in Figure 9.14. The client sends request for service. The message is intercepted by the security context of the client and the username, and the password are inserted into the message before forwarding it to *serverA*. When the message reaches its destination it is intercepted by the security context of the server which enforces authentication and access control. After validating the username and the password, the message is passed to *serverA*.

The scenario illustrated in Figure 9.14 is appropriate when the server trusts the identifier and the proof contained in a message. But the identifier and proof may be captured by a malicious third party and used to obtain unauthorized access to the server. To prevent such attacks, the security context of the server should use a stronger authentication scheme as shown in Figure 9.15. The *client* sends a service request to *serverB*. The security context of the client detects that a `bondCHAPCredential` is used and only forwards the message. The message is captured by the security context of the server. The authenticator of the security context of the server sends a challenge to the credential component of the security context of the client and expects a response derived from both the challenge and information contained in client's credential. The authenticator uses the challenge and corresponding response to authenticate the client. If the service request is validated, the server object grants the service.

9.2 THE AGENTS

The `bond.agents` package implements the agent framework. Our agent model was designed with several objectives in mind:

(i) Assemble dynamically an agent from reusable components. Use a description language to specify the structure of an agent.

(ii) Create a supporting environment for an agent. The environment should be open-ended and support societal services.

(iii) Map the agent description into a data structure and feed this data structure to the control unit responsible for coordinating the execution of an agent.

(iv) Support concurrent activities as a defining feature of an agent rather than an afterthought. Agents should be able to respond promptly to external events and, at the same time, carry out multiple tasks previously initiated.

(v) Support changes in the behavior of an agent. Since behavior is determined by structure, support structural mutations of an agent.

(vi) Support a weaker form of agent mobility, allow agents to migrate at discrete instances of time and to specific locations only. Conceive an architecture where the complexity of the agent state periodically reaches a minimum, and exploit this feature to facilitate mobility. Allow agents to migrate only to sites part of the environment.

Our agent model consists of four components, state machines or planes, strategies, a model of the world, and an agenda. The terms *state machine* and *plane* are used interchangeably throughout this chapter, the first when discussing the agent structure and the second in the context of the functionality of an agent.

Structurally, an agent is a collection of state machine. In turn, each state machine is described by states and transitions amongst states. Strategies, the functional components of an agent are specified for each state.

To describe an agent we introduced an agent description language called *Blueprint*. A Blueprint program is interpreted by an *agent factory* object which creates an internal data structure. In turn, this data structure is used by the agent factory to control the run-time behavior of the agent.

The multi-plane agent model. The agents are described by *functional components* (the strategies) and the *structural components* - the multi-plane state machine. The multi-plane structure provides the means to express concurrent agent activities. Each state machine is said to be operating in its own *plane*, thus the term *multi-plane state machine* for our model. Each plane may performed a different task, one may support reasoning or planning functions, another the execution, while a third one is used for bookkeeping.

A state machine is defined by a graph with nodes corresponding to states and edges to transitions among states. Each state machine has one active node at any given time. The *state of the agent* is defined by a *vector of states*, one state per plane. A state machine changes its state by performing *transitions*. The transitions are triggered by internal or external *events*. External events are messages. The set of external messages that trigger the transitions of one or more state machine defines the *control* *subprotocol* of the agent. Multiple strategies may be used to handle different events e.g. a strategy in one plane for external messages, while another plane handles user interface events.

The behavior of an agent is often *multifaceted*, it consists of several loosely coupled aspects. A full-featured agent may exhibit several facets:

Reasoning. Agents use inference to generate new facts from existing ones using a set of rules.

Visual interface. Most agents present a visual interface and interact with humans: (a) presenting its knowledge i.e. a part of the model in a visual format, and (b) collect user interface events.

Reactive behavior. Agents react to external events.

Active behavior. Agents perform actions in pursuit of their agenda even without external events.

In most cases, a separation of these facets is possible, and the relative independence of the facets justifies their separate treatment. For example the various steps taken by an agent to pursue its goal are changes in its active behavior, but these changes may not necessarily lead to a change in its reactive behavior, the look of the user interface, or the reasoning process of the agent.

The multi-plane state machine structure provides an elegant way to express the multifaceted behavior of an agent, every plane expresses a facet of the behavior of the agent. There are no restrictions on the nature and behavior of planes, so the agent designer can create the structure most suitable to the problem at hand. However, the independence of facets is relative, significant interdependence existing between them. In the multi-plane state machine structure, the interdependence amongst planes is captured by the fact that all planes share a common model and transitions triggered by one plane are applied to the whole structure, providing a signaling mechanism amongst planes.

The strategies are the functional components of an agent. Once a state machine enters a state it triggers the excution of the strategy associated with that state. In turn, a strategy consists of a sequence of `actions` executed under the control of a scheduler.

Strategies are written in programming languages like Java, C, C++, or in interpretative languages like JPython. They can be specified as executables, Java class files, or be embedded in the blueprint as source programs, to be processed by an existing interpreter. The strategies are discussed in depth in Section 9.2.1.3.

The model of the world is an unordered collection of free-formatted items accessed by name, representing all the information an agent has about the environment and itself. The model, can be a knowledge base, an ontology, a pre-trained neural network, a collection of meta-objects, handles of external objects e.g. file handles, sockets, etc., or a heterogeneous collection of all the above. It also contains agent state information.

The model of the world is a Bond object itself, with a set of dynamic properties, one for each component. The model is used by strategies as a shared memory, strategies communicate with each other by storing and retrieving data to/from the model. The naming scheme supports namespaces and allows multiple strategies to reuse variable names without conflicts. Programming languages like C++ use namespaces to resolve name conflicts.

The model of the world is a passive object, inherits the serializability and mobility properties of Bond objects and allows migration and checkpointing of Bond agents. The information in the model might be time and location dependent and be meaningless after migration. For example, the string `/usr/bin/netscape` giving the path information for the executable of a browser is meaningless when an agent migrates from a Linux to a Windows NT system.

The agenda is an object that defines the goal of the agent. The agenda implements a boolean distance function on the model. The boolean function shows if the agent has accomplished its goal. The agenda acts as a termination condition for the agents, except for agents with a *continuous agenda* where their goal is to maintain the agenda as being satisfied. The distance function may be used by the strategies to choose their actions.

9.2.1 Communication and Control. Agent Internals.

An agent can only exist in a supporting environment provided by a resident. Several objects in this environment control the lifecycle of an agent, see Figure 9.16. The `bondAgentFactory`, assembles the agent based upon its blueprint and generates its *Agent Control Subprotocol, ACS* and an agent control structure. The *ACS* allows the agent to communicate with other objects. The agent control structure is an internal data structure used by the `bondSemanticEngine` and the `bondActionScheduler` to control the runtime behavior of the agent.

The structural and functional components of the agent, the blueprint and the strategies come from local or from remote repositories. The agent factory assembles an agent based upon its blueprint and may also create a modified blueprint if the control agent structure is modified at runtime as discussed in Section 9.2.1.9.

Each state of each plane has a strategy associated with it. Strategies may be loaded statically when an agent is created, or dynamically, at the time of a transition to the corresponding state. Strategies may come from the local strategy database, may be downloaded from an Web server, or from the tuple space, or may be provided by the entity requesting the creation of the agent. At the time of this writing, strategies in the JPython scripting language may be included in an agent control message.

All objects, including agents react to messages by invoking methods implemented by the object. To understand the behavior and functions of an object we examine the two facets of an object: (i) message decoding and (ii) the actions taken by the object in response to messages described by the methods supported by the object.

In this section we describe the major events in the life of an agent as follows: creation in Section 9.2.1.5; activation in 9.2.1.6; checkpointing and restarting in 9.2.1.7; migration in 9.2.1.8; modification or surgery in 9.2.1.9. These events occur in response to messages sent either to the agent factory controlling the agent, or to

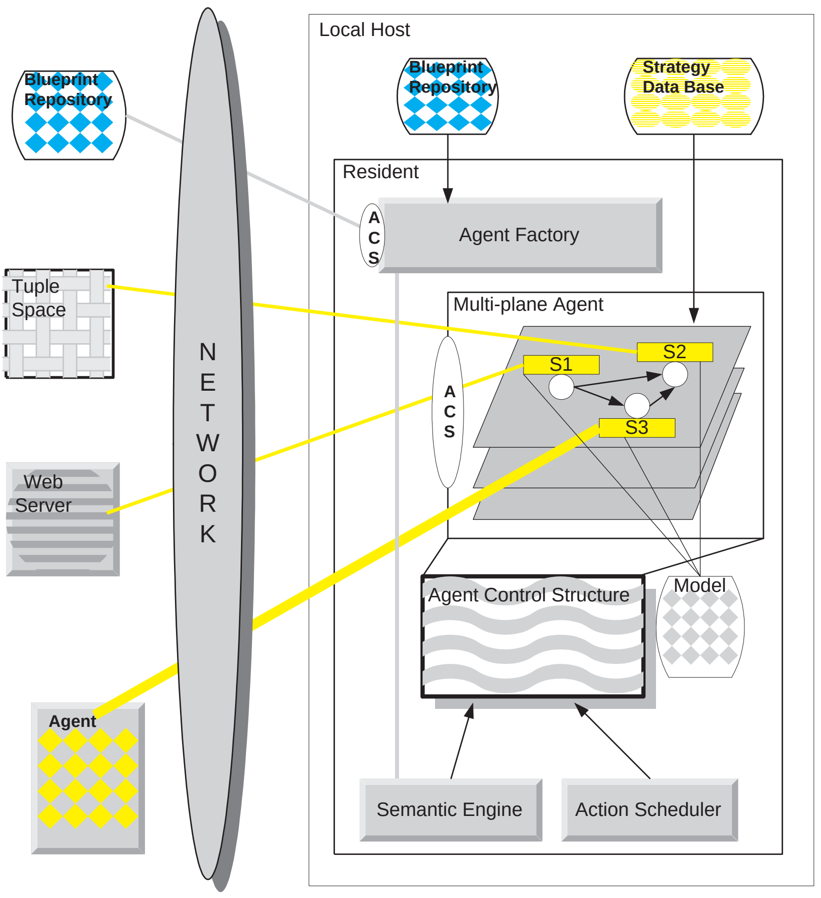


Fig. 9.16 The agent runtime environment. The agent and its *Agent Control Subprotocol, ACS* are created by the `bondAgentFactory`. The structural component of an agent, the *blueprint*, and the functional components, *strategies*, come from local or from remote repositories. The agent has multiple planes, each plane is a state machine. Each state of a state machine has a strategy associated with it. Once created, the `bondActionScheduler` and the `bondSemanticEngine` control the execution of the agent using an internal data structure. Strategies can be loaded dynamically from: local repositories ($S2$), from Web servers ($S1$) or may be written in a scripting language and transmitted in a message from another agent ($S3$). Strategies communicate with one another through the model.

the agent itself. The messages controlling the life cycle of an agent form the agent control subprotocol discussed in Section 9.2.1.1.

9.2.1.1 The Agent Control Subprotocol. An agent uses a dynamically created agent control subprotocol to communicate with: the agent factory; the entity controlling the agent; other objects including agents. The messages of the *agent control subprotocol* are described in Table 9.7. These messages are used to: control checkpoint and restart, modify, and migrate an agent.

The agent control subprotocol follows the major events in the lifetime of the agent; it is created dynamically when the agent is assembled; disappears when the agent is killed; it is modified when the agent undergoes surgery. The agent control subprotocol is itself an object and may be distributed to other objects.

The agent control subprotocol requires actions to be taken by the agent factory or by the agent. The following messages are sent to the agent factory controlling the agent and invoke methods of `bondAgentFactory`: `assemble-agent`, `checkpoint`, `modify-agent`, `migrate-from-here`, `kill-agent`. The agent itself supports methods to communicate with the model `getModel`, `setModel`, to report the state, `getState`, or to provide its subprotocol, `learn-subprotocol`. The methods supported by `bondAgent` are discussed in Section 9.2.1.2, the ones supported by `bondAgentFactory` in 9.2.1.4.

9.2.1.2 Agent Communication. The `bondAgent` has a constructor for an empty agent and methods to start, stop, soft-stop, and kill an agent. The constructor sets up one of the action schedulers and the semantic engine. The round-robin scheduler is the default. Starting and stopping an agent implies starting and respectively stopping the scheduler. Now all agents support the fault detection mechanism (see Section 9.2.4.2), initialized at the time an agent is started.

The `say` method of an agent supports by default delivery of messages in the agent control control and fault detection subprotocols. In addition, it delivers external messages that may cause transitions of the state machines. An external message is delivered to all state machines. Finally, the `say()` method falls back on the the `say` method of the ancestor.

```
public bondAgent() {
   model = new bondModel();
   initStrategyPath();
   String schedulerName = System.getProperty
                                 ("bond.scheduler");
   if (schedulerName.equals("MT")) {
     basched = new bondMTActionScheduler(this);}
   else {
     if (!schedulerName.equals("RR")) {
        Log.Debug("Action scheduler invalid, using RR"); }
        basched = new bondRRActionScheduler(this);}
   semantic = new bondStateMachineSemantic(planes, model);
 }
public void say(bondMessage m, bondObject sender)
   try {
     if (m.getSubprotocol().equals("AgentControl")) {
       sphAgentControl(m, sender); return;}
     if (m.getSubprotocol().equals("FaultDetection")){
      sphFaultDetection(m, sender); return;}
```

Table 9.7 The messages of the agent control subprotocol. The entities involved are: the *beneficiary*, the *agent*, the *agent factory controlling the agent*, AgF, the *agent factory at a new location*, AgF_{new}.

Message	Parameters	Message function
`assemble-agent`	`:blueprint` `:blueprint-address` `:visual`	Sent to AgF and request to assemble an agent using the blueprint downloaded from `blueprint-address`. Specify `:visual` if editor window is desired.
`agent-created`	`:bondID` `:bondAddress`	Sent to beneficiary by AgF to confirm creation of agent. Gives `bondID` and `bondAddress`.
`start-agent`	`:model` `:alias`	Sent to agent by beneficiary. Request agent to start or resume execution.
`soft-stop`		Request agent to soft stop.
`checkpoint`	`:bondID` `:checkpoint file`	Sent to AgF. Agent factory soft stops agent `:bondID`, saves its current state to local file `:checkpointfile`, and restarts agent.
`checkback`	`:bondID` `:checkpoint file`	Sent to AgF. Agent factory soft stops the agent, restores its state from local file `:checkpointfile` and restarts agent.
`modify-agent`	`:blueprint` `:blueprint-address`	Sent to AgF. Request to modify the agent. Surgical blueprint embedded in the `blueprint` or downloaded from `blueprint-address`.
`migrate-agent`	`:blueprint` `:visual` `:bondID` `:modelID`	Sent to AF_{new} by AgF. AF_{new} re-creates the agent with `:bondID` using the embedded blueprint and realizes the model of agent, `:modelID` from the source site.
`migrate-from-here`	`:bondID` `:remote-address`	Sent to AgF. Initializes migration of agent `:bondID` from source to destination site `:remote-address`.
`migrated`	`:bondID`	Sent to AgF by AgF_{new}. Successful migration. Request AgF delete old copy of agent.
`kill-agent`	`:bondID`	Sent to AgF . If running, agent is soft-stopped and disposed of.
`getModel`	`:property`	Sent to agent. Agent replies with the value of the property `property` from the model.
`setModel`	`:property` `:value`	Sent to agent. Sets value of the model property `:property` to `:value`
`getState`		Sent to agent. Agent responds with its current state vector.
`learn-subprotocol`		Request agent to generate and send subprotocol object.

```
      if (genericSPH(m, sender)) {return;}
      if (m.getSubprotocol().equals(sp.getName())) {
         for(Enumeration e=planes.elements();
                           e.hasMoreElements(); ) {
             bondAgentPlane bap = (bondAgentPlane)
                                e.nextElement();
             bap.fsm.say(m,sender);}
      } else {
    super.say(m, sender);}
    }
    catch (NullPointerException e) { }
 }
```

The code for the agent control subprotocol listed below handles the following messages: `get-state`, `start-agent`, `stop-agent`, `kill-agent`, `getModel`, `setModel`

```
public void sphAgentControl(bondMessage m, bondObject sender) {
  if (sender == null) sender = m.getSender();
  if (restricted_control && !sender.equals(beneficiary)){
      sender.say( m.createReply("(deny)"),this); }
  if (m.content.equals("get-state")) {
      String state = "";
      for(Enumeration e=planes.elements();e.hasMoreElements();){
      bondAgentPlane bap = (bondAgentPlane) e.nextElement();
      state += "."+bap.fsm.getState().getName();}
      sender.say( m.createReply("(tell :content state:
                            state "+state+")"),this);}
  if (m.content.equals("start-agent")) {
      initDropBox();
      populateModel(m.getParameter(":model"));
      String als = (String)m.getParameter(":alias");
      if (als != null) dir.addAlias(als, this);
      initFaultDetection();
      start();
      sender.say( m.createReply("(tell :content ok)"),this);
      return;}
  if (m.content.equals("stop-agent")) {
      softstop = true;
      sender.say( m.createReply("(tell :content ok)"),this);
      return;}
 if (m.content.equals("kill-agent")) {
      kill();
      sender.say( m.createReply("(tell :content ok)"),this);
      return;}
 if (m.content.equals("getModel")) {
      Object val = model.get((String)m.getParameter
                                      (":property"));
      bondMessage rep = m.createReply("(tell :content value)");
      rep.setParameter(":value", val);
      sender.say(rep,this);
      return;}
 if (m.content.equals("setModel")) {
      model.set((String)m.getParameter(":property"),
                                  m.getParameter(":value"));
      if (m.getParameter(":createReply") != null)
                 {m.sendReply("(tell :content ok)", this); }
      return;}
}
```

9.2.1.3 Strategies. Strategies are the functional components of an agent. Formally, a strategy is a function which takes as parameters the model of the world and the agenda of the agent and returns actions. A strategy implements three interfaces, `install()`, `action()`, and `unistall()`, see Figure 9.17. When the state machine generates a transition to a state the thread of control of invokes the three methods in this order.

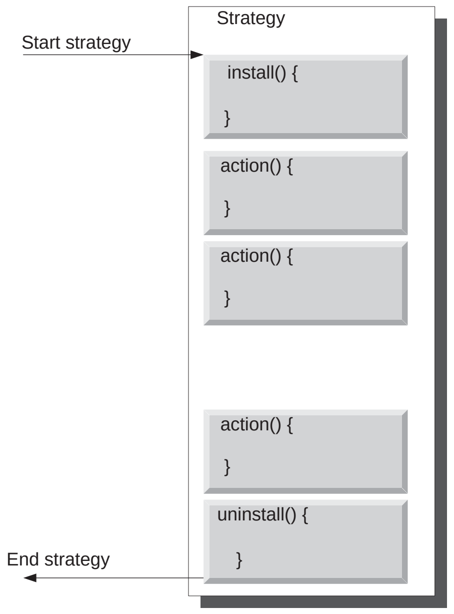


Fig. 9.17 The structure of a strategy.

The actions determine the behavior of the agent. Actions are atomic and strategies to not reveal their entire state to the agent or the environment. While a strategy executes, it cannot be interrupted and its state may be rather complex.

A strategy consists of a sequence of actions, in an infinite sequence interrupted only when a transition takes place. An alternative approach is to have *one-shot strategies*, generating only one action, followed by a transition. A strategies is activated as the flow of control requires, or in response to external events.

Messages from remote applications, and *user interface events* like pressed keys, mouse-clicks are examples of external events. The strategies are activated by the event handling mechanism - the Java event system for GUI events, or the messaging thread for messages in case of external events, or by an *action scheduler*. Activation using the external messages is characteristic for strategies derived from `bondProbeStrategy`, while activation as a result of user interface events are handled by strategies derived from `bondGuiStrategy`.

The model is used by strategies as a shared memory, strategies communicate with each other by storing and retrieving data to/from the model. The naming scheme supports namespaces and allows multiple strategies to reuse variable names without conflicts. Programming languages like C++ use namespaces to resolve name conflicts.

There are two methods `getModel` and `setModel` to read and write data into the model. By default, a strategy accesses only its own namespace but may address variables outside its namespace by specifying the full name of the variable. The default namespace of a strategy is specified in the Blueprint of the agent.

Example. The blueprint statement:

```
add state ExecBrowser with strategy Exec.Start::Browser;
```

means that the `ExecBrowser` strategy uses the namespace `Browser`.

```
String toexec = getModel("commandline");
```

returns the model variable named `Browser.commandline` and

```
setModel("output", commandOutput);
```

writes `commandOutput` into the model variable named `Browser.output` if the methods are invoked by strategies with `Browser` as default namespace.

Only a small fraction of the internal state of a strategy is exposed to the outside world through the model. When the agent enters a new state the strategy associated with that state is activated and may read from the model the current value of model variables. A strategy may deposit results in the model just before completion.

At any given time t the *internal state of an agent* is given by the internal state of all its strategies and the model, the *external state* or the *agent state* is a vector stored in the model describing the state of each state machine.

From the implementation point of view, a strategy is a Java interface with a function called `action()` that performs the actions required when an agent enters a state. The system provides three primitive strategies:

`bondGuiStrategy` handles a GUI window. Initializes the user interface when the strategy is entered, and closes the window upon termination.

`bondProbeStrategy` automatically installs and un-installs itself as a probe for a specific subprotocol.

`bondDefaultStrategy` is a place holder for a real strategy in case of lazy-loading.

The system supports strategies written in: Java, other programming languages wrapped in the Java Native Interface, and scripting languages such as JPython. The following objects may be used as strategies:

Table 9.8 Strategy groups in the strategy database

Name	Function of the strategy group
`Util`	Utility e.g. delay.
`Agent`	Checkpoint, migration, surgery, termination.
`Dialog`	Dialog boxes for: warnings, messages, and yes/no questions.
`Exec`	Start, supervise, and control local applications.
`RemoteExec`	Start, supervise, and manage remote applications.
`AgentExec`	Start and control agents and groups of agents
`FTP`	Data migration.
`Model`	Save, load and merge models.
`Scheduler`	Metaprogram scheduling algorithms.
`Synch`	Strategies for agent synchronization.

(i) Objects derived from `bondDefaultStrategy`, `bondGUIStrategy` or from `bondProbeStrategy`. This is the method of choice to create Java strategies.

(ii) Objects implementing the `bondStrategy` interface. This method allows us to create strategies that inherit from classes outside the Bond hierarchy.

(iii) External objects with Java Native Interface, JNI, wrappers. Any external object written in a programming language other than Java can be transformed into a Bond strategy using a JNI wrapper. The wrapper must implement the `bondStrategy` interface.

(iii) Embedded languages. The source code of a strategy can be embedded into the blueprint specification of an agent. The code can be in an interpreted languages with an existing Java interpreter. We currently support Python, through the JPython interpreter [27] and Clips, in its Jess Java-based incarnation [20].

Most of the strategies in the Bond strategy database grouped together into strategy groups. Table 9.8 lists the most important strategies groups.

9.2.1.4 Agent Factory. The agent factory translates a blueprint agent description into an internal data structure, called agent control structure and then uses this data structure to control the agent as seen in Figure 9.18. An agent may be altered dynamically as discussed in Section 9.2.1.9 and then the agent factory is able to generate a modified blueprint.

The sequence of steps taken by the agent factory to create an agent is:

(i) Get the blueprint and the components.

(ii) Generate the finite state machines and link each state with its corresponding strategy.

(iii) Generate the control subprotocol of the agent.

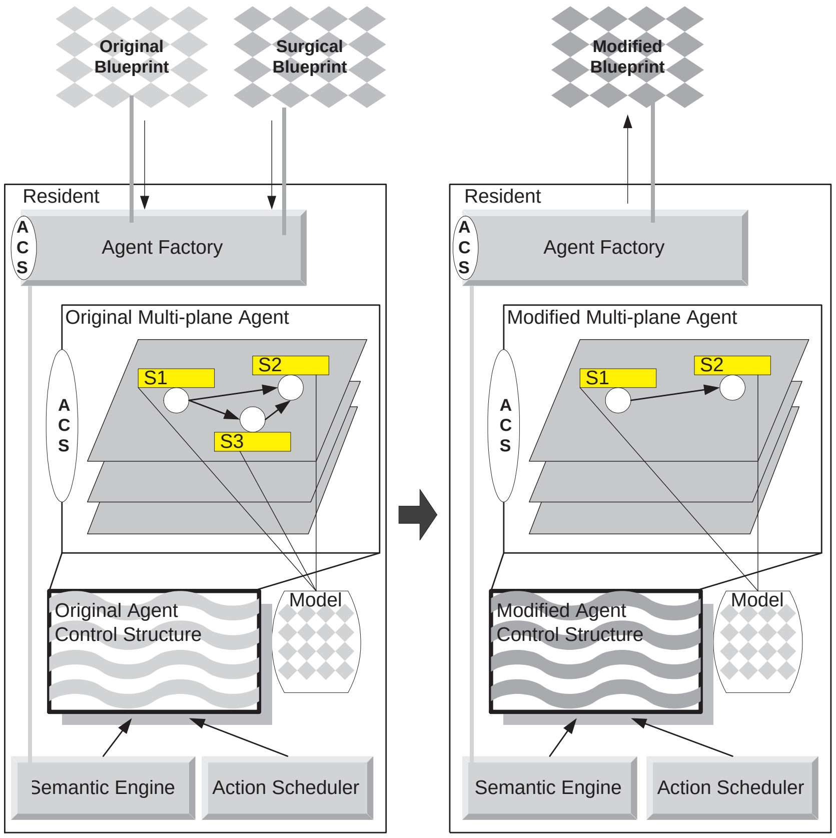


Fig. 9.18 The *Agent Factory* translates a *Blueprint* into an internal control structure and an agent. When a *Surgical Blueprint* is provided, the agent factory modifies the internal data structure controlling the agent and is able to automatically generate the modified blueprint.

(iv) Send a copy of the control subprotocol object to the beneficiary and to other objects the agent need to communicate with .controlling authority, be it a user interface or another agent.

The agent factory controls the runtime behavior of an agent and uses the action scheduler to transfer control to a new action whenever the current one completes its execution. Once a transition from the current state to the next state takes place, the agent factory is responsible to load the strategy corresponding to the new state.

The strategy loader, looks up a strategy, `Foo`, in the following order:

(i) Searches the strategy data bases for the Java class `bondFooStrategy.class`.

(ii) Searches the directories specified in the `import` statements in the blueprint description of the agent. The order of the import statements is important.

(iii) As a last resort considers the strategy name a full name of the Java class, i.e. `Foo.class` and repeats the search in the same order.

After loading the blueprint file, the agent factory parses the script and assembles the agent according to the specification. The initialization of strategies can be done in two modes:

Full-load mode. The strategies are loaded and instantiated at the time the agent is created by the `bondStrategyLoader` object.

Lazy-load mode. None of the strategies are loaded, but they are replaced with a lightweight object called `bondLazyLoadingStrategy`. Whenever a state is entered, the lazy loading strategy attached to the state, triggers the loading of the real strategy, and replaces itself with the real one.

Lazy-loading leads to faster startup time. Moreover, agents with a complex structure, may never reach some of their states, thus the corresponding strategies will never be entered. On the other hand, the loading process triggered by entering a state will cause delays during the execution, thus this method is not suitable for agents operating in a real-time environment.

Mobile agents may travel to sites where some of the strategies are not available. In this case, the lazy loading may prevent some load-time errors. When an agent migrates to a new site, each strategy is loaded again when the agent enters the corresponding state. In this case a different strategy, the one available locally will be loaded instead of the strategy used at the original site. This feature can be used to customize an agent depending upon the current host. For example, when an agent migrates to a palmtop computer a different user interface than the one for a desktop may be used.

The lazy strategy loading differs in scope and implementation from the run-time linking provided by the Java class loader. Java loads classes at their first instantiation and the linker assumes that the class was known at compile time, although it can be cheated into loading classes it had never seen before. This just-in-time loading is especially useful for applets, because it helps hiding the network latency and provides for a faster startup.

The `bondAgentFactory` is an object with the alias "AgentFactory" that implements some of the methods for the agent control subprotocol. The methods are describe in the following sections when we discuss the milestones in the life cycle of an agent, `assemble-agent` in Section 9.2.1.5, `checkpoint` and `checkback` in Section 9.2.1.7 and `migrate-agent`, `migrate-from-here`, and `migrated` in Section 9.2.1.8. Here we only present the code for the agent control subprotocol. Once the contents of a message is identified the corresponding method of the agent factory is invoked.

```
public class bondAgentFactory
  extends bondProbe {
    public bondAgentFactory() {
    dir.addAlias("AgentFactory", this);
  }
 public void say(bondMessage m, bondObject sender) {
    if (genericSPH(m, sender)) { return;}
    super.say(m, sender);}
 public void sphAgentControl(bondMessage m, bondObject sender) {
    if (m.content.equals("assemble-agent")) {
      assembleAgent(m, sender);}
    if (m.content.equals("modify-agent")) {
      modifyAgent(m, sender);}
    if (m.content.equals("migrate-agent")) {
      migrateAgent(m, sender);}
    if (m.content.equals("migrate-from-here")) {
      migrateFromHere(m, sender);}
    if (m.content.equals("migrated")) {
      migrated(m, sender);}
    if (m.content.equals("checkpoint")) {
      checkpoint(m, sender); }
    if (m.content.equals("checkback")) {
      checkback(m, sender);}
    if (m.content.equals("kill")) {
      kill(m, sender);}
}
```

9.2.1.5 Agent Creation. The agent creation process is triggered when the agent factory receives an `agent-create` message. This message can be: (i) sent by another object, (ii) generated locally by the `RunAgent` object from command line parameters, or, (iii) generated by the user from a local or remote agent control panel. Then the agent factory method `assembleAgent` is invoked.

```
 void assembleAgent(bondMessage m, bondObject sender) {
    String visual = (String)m.getParameter(":visual");
    bondAgent ba = interpretFromMessage(m, null);
    if (ba == null) {
      m.sendReply("(error :content BadBlueprint)", this);
      return; };
    String res = (String)m.getParameter(":repository");
    if (res != null)
      System.setProperty
             ("bond.current.strategy.repository", res);
    if (sender instanceof bondShadow) { ba.beneficiary =
                                    (bondShadow)sender;}
    else { }
    if (visual == null) {
       String visualFlag = System.getProperty
                                ("bond.agent.visual");
       if (visualFlag != null && visualFlag.equals("true"))
           visual = "yes";
       else   visual = "no";
    }
    if (visual.equals("yes")) {ba.edit();}
    bondMessage rep = m.createReply
                      ("(tell :content agent-created)");
    rep.setParameter(":bondID", ba.bondID);
    rep.setParameter(":address",
                       com.localaddress+":"+com.localport);
    if (!(sender instanceof bondShadow)) {
       bondShadow t = new bondShadow(sender);
       t.say(rep, this);}
    else { sender.say(rep,this);}
 }
```

The blueprint for the new agent may be provided within the message or may be specified using the `repository` parameter of the message. An agent may be created with or without a visual editor.

The object sending the `agent-create` request to an agent factory is called the *beneficiary* of the agent. There is a special relationship between an agent and its beneficiary. The agent keeps a shadow of its beneficiary and sends notifications regarding important events in its lifetime such as: termination, migration, error conditions.

After the agent is successfully created, the agent factory sends the `agent-created` message to the beneficiary. However, the agent is not started immediately after its creation. The beneficiary may initialize the model between creation and the agent start.

The beneficiary may request the agent to reject messages from other objects and communicate exclusively with itself. This is done by setting the `beneficiary-only` parameter in the `agent-create` message. This security mechanism is similar with the sandbox security model of Java [39], [22].

The`interpretBlueprint` method of the `bondAgentFactory` invokes a blueprint parser and examines one of the switches of the configuration file to determine if lazy loading is in effect.

```
 public bondAgent interpretBlueprint(Reader is,
                                         bondAgent ba) {
    bond.agent.blueprint.syntaxtree.Node root = null;
    blueprintParser parser = new blueprintParser(is);
    if (parser == null) return null;
    try {
      root = parser.BluePrintProgram();}
    catch (ParseException pex) { }
    if (root == null) return null;
    BlueprintInterpreter bp = new BlueprintInterpreter();
    bp.lazyLoad = Boolean.getBoolean("bond.agentLazyLoading");
    bp.ag = ba;
    root.accept(bp);
    return bp.ag;
  }
```

The`interpreFromMessage` method of the `bondAgentFactory` determines if the blueprint is supplied with the message by examining the `blueprint-program` parameter and if so invokes the blueprint parser.

```
bondAgent interpretFromMessage(bondMessage m,
                                           bondAgent ba){
   bondEmbeddedBlueprint blueprint_prog =
     (bondEmbeddedBlueprint)m.getParameter
                                  (":blueprint-program");
   if (blueprint_prog != null) {
     return interpretBlueprint(
                         blueprint_prog.getReader(),ba);}
   String blueprint = (String)m.getParameter (":blueprint");
   if (blueprint != null) {
     return interpretBlueprint(
                           openBlueprint(blueprint), ba);}
   bondEmbeddedBlueprint xml_blueprint_prog =
     (bondEmbeddedBlueprint)m.getParameter
                               (":xml-blueprint-program");
   if (blueprint_prog != null) {
     return interpretXMLBlueprint
                     (xml_blueprint_prog.getReader(),ba);}
   String xml_blueprint =
                 (String)m.getParameter(":xml-blueprint");
   if (xml_blueprint != null) {
     Reader is = openBlueprint(xml_blueprint);
     return interpretXMLBlueprint(is, ba);}
   return null;
}
public Reader openBlueprint(String bpfile) {
   if (bpfile.startsWith("http://")) {
     try{
       URL con = new URL(bpfile);
       return new InputStreamReader(con.openStream());}
     catch (MalformedURLException muex) { }
     catch (IOException ioex) { }
   } else {
     try { return new FileReader(bpfile);}
     catch (FileNotFoundException fnfex) { }
     }
}
```

9.2.1.6 Agent Activation. The `start-agent` message triggers the activation of the agent. The processing of this message is illustrated by the code presented in Section 9.2.1.2. Upon receipt of this message:

(i) if the message includes the `:model` parameter then then model is initialized by the `populateModel` function listed below.

(ii) the state vector of the multi-plane state machine becomes the initial state specified in the blueprint,

(iii) the current strategies are installed,

(iv) the execution thread is created, and

(v) the *action scheduler* starts to execute actions according to the current strategies.

```
public boolean populateModel(Object mXML) {
  if (mXML == null) return false;
  bondXMLmodel temp = new bondXMLmodel();
  temp.setModel(model);
  if (mXML instanceof bondEmbeddedBlueprint) {
    bondEmbeddedBlueprint model_XML =
                  (bondEmbeddedBlueprint)mXML;
    temp.fromXML(model_XML.getReader());}
  else {
    String model_XML = (String)mXML;
    if (model_XML.startsWith("http://") ||
      model_XML.startsWith("HTTP://") ||
      model_XML.startsWith("file:/") ||
      model_XML.startsWith("FILE:/")) {
        temp.fromXML(model_XML);}
    else { temp.fromXML(new ByteArrayInputStream
                        (model_XML.getBytes()));}
  }
  return true;
}
```

In the default running mode the active strategies of the agent perform actions. These actions are performed in response to:

- action scheduler polling,

- user interactions handled by GUI strategies, and

- external messages handled by probe strategies.

The vector of currently active strategies can be changed as a result of transitions. Transitions are triggered as a result of messages. These messages can be sent either from the current strategies of the agent (*internal transitions*) or from external objects (*external transitions*). The internal transitions for a special group in the blueprint specification, and they represent events which are intrinsically linked to the currently active strategy like success or failure. The agent framework does not allow external objects to trigger internal transitions. External transitions correspond to commands, and they can be triggered both externally or internally.

The execution of Bond agents can be stopped with the `stop-agent` message. This message instructs the action scheduler to stop the execution of the agents on the next *action boundary*. Thus a soft stop is not instantanenous, and the time until it occurs depends upon the action scheduler (single threaded or multi-threaded) and on the granularity of the actions. At a soft stop of an agent the message handling is blocked, so the strategies triggered by messages or user input are blocked too.

9.2.1.7 Agent Checkpoint and Restart. In a soft stopped state, the current status of the agent can be checkpointed. This is done by sending the `checkpoint` message to the agent factory. The agent factory will serialize the model of the agent to a file indicated in the `:file` parameter of the message. The agent editor window of the agent allows interactive checkpointing.

The reverse operation of checkpointing is the checkback operation, triggered by a `checkback` message sent to the agent factory. The agent factory performs soft stops the agent if it is running, restores the model, and reinstalls the state vector to the strategies which were active at the moment when the agent was checkpointed.

The `bondAgentFactory` has two methods to support checkpointing and restarting an agent. The first method extracts the unique `agentid` and the name of the checkpoint file. Then it locates the agent and then calls the `writeObject` method. As a result, a copy of the agent model is written into the checkpoint file.

```
void checkpoint(bondMessage m, bondObject sender) {
   String agentid = (String)m.getParameter(":agentid");
   bondAgent ag = (bondAgent)dir.findLocal(agentid);
   if (ag == null) { return;}
   try {
     String checkpointfile = (String)m.getParameter
                                (":checkpointfile");
     FileOutputStream fs = new FileOutputStream
                                   (checkpointfile);
     ObjectOutputStream outs = new ObjectOutputStream(fs);
     outs.writeObject(ag.model);
     outs.close();
   } catch(IOException ioex) {}
}
void checkback(bondMessage m, bondObject sender) {
   String agentid = (String)m.getParameter(":agentid");
   bondAgent ag = (bondAgent)dir.findLocal(agentid);
   if (ag == null) { return;}
   try {
     String checkpointfile = (String)m.getParameter
                                (":checkpointfile");
     FileInputStream fs = new FileInputStream
                                   (checkpointfile);
     ObjectInputStream outs = new ObjectInputStream(fs);
     if (ag.running) {
         ag.softStop();
         ag.model = (bondModel)outs.readObject();
         setStatus(ag);
         ag.start();}
     outs.close();}
   catch(IOException ioex) { }
   catch(ClassNotFoundException cnfex) { }
}
```

9.2.1.8 Agent Migration. The system implements *weak migration* of agents, they are allowed to migrate only when all strategies active at the time of the request have completed their execution. At that moment no threads are running, all strategies

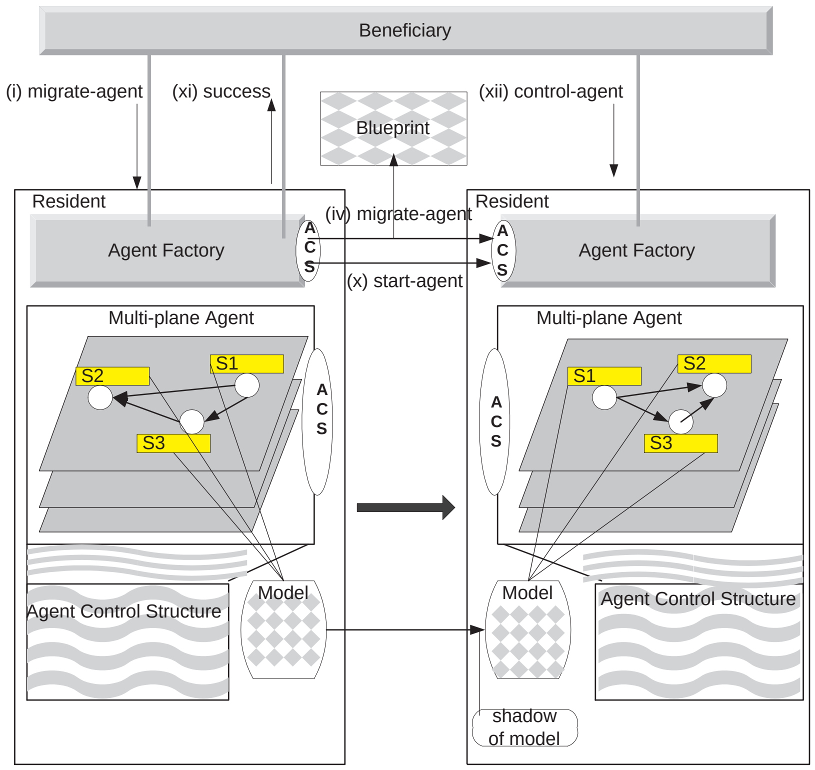


Fig. 9.19 Messages exchanged during agent migration.

have completed their execution and the state of the agent is minimal, it coincides with the state vector of the agent.

This approach reflects the view that migration is a relatively rare event in the life of agents. It also reflects the difficulties of migrating running Java programs. Java does not support thread migration, thus to migrate a running Java program all running threads must be stopped, their status saved, and then recreated at the destination site.

To migrate an agent we have to send to the new site its blueprint and model. The blueprint and the model are passive objects, one is an ASCII file and the other a data structure and their serialization is fully supported by Java.

The migration process involves the agent factory controlling the agent, AgF, and the one at the new resident, AgF_{new}, and consists of the following sequence of the events:

(i) The migration process is initiated by a `migrate-agent` message sent to AgF. The message contains the address of AgF_{new} and the `bondID` of the agent.

(ii) AgF soft stops the agent.

(iii) AgF generates the blueprint of the agent using the internal data structure reflecting the current agent state. This structure may be different than the original agent structure. The mapping is done by the `bondAgentToBlueprint` class.

(iv) AgF sends to AgF_{new} the blueprint generated in step (iii), embedded into a `migrate-agent` message.

(v) AgF_{new} re-assembles the agent from the blueprint. The new agent is a copy of the old one, but it does not have the model yet.

(vi) AgF_{new} creates a shadow of the model of the original agent, and realizes it. The model is thus transferred to the new host.

(vii) AgF_{new} calls the `relocate()` function on the model.

(viii) AgF_{new} sends to AgF a `migrated` message to report the successful creation of the agent.

(ix) AgF un-registers the old agent and makes it eligible for garbage collection. It also installs a *forwarder* object if the `:forwarder yes` parameter was specified. This object forwards any messages sent to the agent at the old site to its new location.

(x) AgF sends sends a `start-agent` message to the agent at its the new location.

(xi) AgF sends a `success` message to the originator.

(xii) The beneficiary sends agent control messages to AgF_{new}.

A successful migration requires that the information in the model be moved to another site. Information like handles to open files are meaningful only locally. A set of rules must be observed to make the model mobile - for example, keeping all immovable information inside atomic actions. This implies that we should open and close a file inside a single action.

The `bondMigrationStrategy` allows an agent to trigger its own migration to a new location. The target of the migration process may be specified by a model variable and the decision to initiate the migration can be based on a pre-defined conditions, or due to situations detected by other strategies of the agent, possibly from a different plane. An external agent may decide the target location and the time of migration. For example a controller agent can relocate a set of agents to sites where they are needed. Agent migration can also be done using the user interface, locally from the agent editor, or remotely using the remote agent control panel.

The `bondAgentFactory` methods for agent migration are presented now.

```
void migrateAgent(bondMessage m, bondObject sender) {
   String agentid = (String)m.getParameter(":agentid");
   if (sender == null) sender = m.getSender();
   String visual = (String)m.getParameter(":visual");
   bondIPAddress address = ((bondShadow)sender).remote_address;
   bondAgent ba = interpretFromMessage(m, null);
   if (ba == null) {
     m.sendReply("(error :content BadBlueprint)", this);
     return; };
   String modelid = (String)m.getParameter(":modelid");
   bondShadow shModel = new bondShadow(modelid, address);
   ba.model = (bondModel)shModel.realize();
   if (visual==null || visual.equals("yes")) {
     ba.edit(); }
   setStatus(ba);
   ba.start();
   bondMessage rep = new bondMessage("(tell :content
                               migrated)","AgentControl");
   rep.setParameter(":agentid", agentid);
   sender.say(rep,this);
}
void migrateFromHere(bondMessage m, bondObject sender) {
   // find the local agents
   String agentid = (String)m.getParameter(":agentid");
   bondAgent ag = (bondAgent)dir.findLocal(agentid);
   if (ag == null) {return;}
   String remoteAddress = (String)m.getParameter
                                       (":remote-address");
   ag.softStop();
   bondShadow shFactorynew = new bondShadow("Resident",
                                            remoteAddress);
   bondMessage mes = new bondMessage(
           "(tell :content migrate-agent)","AgentControl");
   bondEmbeddedBlueprint ebp = new bondEmbeddedBlueprint();
   bondAgentToBlueprint a2b = new bondAgentToBlueprint(ag);
   a2b.generate();
   ebp.value = a2b.toString();
   mes.setParameter(":blueprint-program", ebp);
   mes.setParameter(":modelid",ag.model.bondID);
   mes.setParameter(":agentid",ag.bondID);
   shFactorynew.say(mes,this);
}
void migrated(bondMessage m, bondObject sender) {
   String agentid = (String)m.getParameter(":agentid");
   bondAgent ag = (bondAgent)dir.findLocal(agentid);
   if (ag == null) { return;}
   bondEditor ed = (bondEditor)ag.get("Editor");
   if (ed != null) {ed.close();}
   dir.unregister(ag);
}
```

9.2.1.9 Agent Surgery. The dynamic modification of the structural components of an agent is called *agent surgery*. The changes are described by a *surgical blueprint script*. Surgical scripts act on existing agents, and may contain `delete` and `replace` operators. The format of surgical blueprint scripts are described in detail by the BNF syntax specification.

The agent surgery is triggered by the `modify-agent` message sent by an object to the agent factory controlling the agent. The sequence of actions in this process is:

(i) A *transition freeze* is installed. The agent continues to execute normally, but if a transition occurs the corresponding plane is frozen. The transition will be enqueued, and executed when the transition freeze is lifted.

(ii) The agent factory interprets the blueprint script and modifies the multi-plane state machine accordingly. Two special cases are considered:

(ii.a) If an entire plane is deleted, the plane is brought first to a soft stop - i.e. the last action completes.

(ii.b) If the current node in a plane is deleted, a *failure* message is sent to the current plane. If there is no failure transition from the current state, the new state will be a null state. This means that the plane is disabled and will no longer participate in the generation of actions. and executed when the transition freeze is lifted.

(iii) The transition freeze is lifted, the pending transitions performed, and the modified agent continues its existence.

An agent may initiate the surgical operation itself using the `bondAgentSurgery` strategy. This strategy takes the address of the surgical blueprint script from the model. The surgery may be initiated by a remote agent or may be triggered by a user from an agent control panel.

The surgery is useful to build up a sophisticated agent capable to perform complex actions from a simple generic agent. For example in a network discovery application, a simple discovery agent is sent to a remote site by a controller agent. As the discovery agent learns more about the remote environment it is upgraded using a sequence of surgical blueprints sent by the controller agent.

The `modifyAgent` method of the `bondAgent` is listed below.

```
void modifyAgent(bondMessage m, bondObject sender) {
   bondAgent ba = null;
   String agentid = (String)m.getParameter(":agentid");
   ba = (bondAgent)dir.findLocal(agentid);
   if (ba.running) {
     ba.softStop();
     ba = interpretFromMessage(m, ba);
     ba.start();}
   else { ba = interpretFromMessage(m, ba);}
   if (ba == null) {m.sendReply("(tell
            :content error agent not modified)",this);}
   else {
     m.sendReply("(tell :content agent-modified)",this);}
}
```

9.2.1.10 Action Scheduler.

The action scheduler transfer control to an action at the time of a transition or at the completion of the current action. Actions are are the primitives used by a strategy to accomplish its functions. An action notifies the scheduler upon completion.

At this time we have two scheduler objects. Both schedulers guarantee that actions from the same strategy do not overlap.

(i) The `bondRRScheduler` supports a single-threaded, round-robin scheduling of actions across state machines.

(ii) The multi-threaded action scheduler `bondMTScheduler` allows multiple actions from different planes to be executed concurrently.

The `bondRRScheduler` identifies the state of a state machine in one plane and schedules for execution the action associated with the strategy of the current state. When the action finishes it notifies the scheduler, the state of the plane is updated and the scheduler moves to the next plane. The process continues, one action at a time. The scheduler may activate a strategy in response to an event as soon as the current action finishes. For example, a strategy may inform the scheduler that it may not take any action for a specific time and provide the *expected next action* time. This allows the action scheduler to skip the activation of the strategy during the normal round robin activation but it will activate the strategy once the timeout expires. This scheduling strategy assumes that a strategy is decomposed into a set of short actions.

The `bondMTScheduler` iterates over the set of planes, in each plane it identifies the current state, starts up a new thread, and then waits to be interrupted by a notification from any of the threads currently running actions. When a thread is started it identifies the state and the strategy and runs the code of the action. When the action terminates it notifies the scheduler.

```
/* Run a strategy in the context of this thread */
public void run() {
   setRunning(true);
   boolean firstTime = true;
   while ((ba.agenda == null) ||
           !ba.agenda.satisfiedBy(ba.model)) {
       bondStrategy strat =
              ap.fsm.getState().getStrategy();
       if (softstop) { sched.decr(); return;}
       if (strat != null) {
           strat.action(ba.model, ba.agenda);}
       if (!firstTime) {
           try { sleep(500);}
           catch (InterruptedException e) {}
       }
       firstTime = false;
   }
   setRunning(false);
}
/** Start thread */
public void start() {
   softstop = false;
   AgentThread = new Thread(this);
   AgentThread.start();
}
/** Main agent loop */
public synchronized void run () {
    if (ToKill) {  ba.kill();  return;}
    for (Iterator i = ba.planes.iterator(); i.hasNext(); ) {
        bondAgentPlane ap = (bondAgentPlane)i.next();
        bondFiniteStateMachine fsm = ap.fsm;
        fsm.setState(fsm.getState());
        PlaneThread thr = new PlaneThread(this, ap);
        synchronized (threads) { threads.put(fsm, thr);}
        thr.start();
    }
    count = threads.size();
    while (count > 0) {
        try { wait(); }
        catch (InterruptedException e) { }
    }
    for (Enumeration e=ba.planes.elements();
                                    e.hasMoreElements(); ) {
        bondAgentPlane ap = ((bondAgentPlane)e.nextElement());
        bondStrategy strat = ap.fsm.getState().getStrategy();
        if (strat != null) { strat.uninstall();}
    }
 }
}
```

9.2.1.11 Semantic Engine. A semantic engine controls the transition from one state to another. Semantic engines objects can be changed without the need to recompile the agents. At the time of this writing we have only a default semantic engine but more sophisticated ones are under discussion.

The default semantic engine: (a) supports only unconditional transitions, (b) the actions are associated with the nodes of the state machines, once a state machine enters a certain state the strategy associated with that state is activated, and (c) executes the transitions immediately upon receiving events. It discards the events if they do not correspond to a valid transition at the instance they arrive. The operation of the default semantics engine is summarized by the following pseudocode:

```
forall (incoming message m)
  if message is transitionAll t
    forall (planes p)
      if transition t exist from current state on plane p
         call uninstall on current strategy
         change state to the endpoint of transition t
         call install on current strategy
      else
         ignore
      endif
    discard message
  else if message is transition t on plane p1
    if plane p1 exists
      if transition t exist from current state on plane p1
```

```
         call uninstall on current strategy
         change state to the endpoint of transition t
         call install on current strategy
      endif
    endif
    discard message
  endif
endfor
```

More sophisticated semantic engines could support:

Conditional transitions. The conditions should be specified as metadata attached to the multiplane state machine structure.

Buffering of events. A semantic engine could buffer events, and apply them at a later time.

Actions associated with transitions. The default semantic can be extended allowing actions executed whenever the transition is triggered.

Synchronization rules amongst planes.

The statecharts model as described in Harel et. al [25] uses conditional transitions and actions associated with transitions.

9.2.2 Agent Description

9.2.2.1 The Blueprint. We use an agent description language called *Blueprint* to specify the structural components and to initialize the model of an agent. The BNF syntax of Blueprint is presented elsewhere [2]. A blueprint is designed by a programmer and can also be generated by the `AgentFactory` object, see Section 9.2.1.4. A blueprint agent description is a text file, it can be easily transported over the network, embedded in a message or downloaded from web servers.

The agent description starts with `import` statements. The `create agent` and `end create` declarations mark the beginning and the end of the agent description. An agent description consists of several planes. Whenever a statement like `plane foo` is encountered, the agent factory searches the component databases for a plane named `foo` and creates a new plane if the search fails. If the search is successful the plane is opened and subsequent declarations may add new components to the existing structure.

Plane descriptions consists of description of states, as well as internal, and external transitions. The statement:

```
add state StateName with strategy StrategyName;
```

declares a state called `StateName` with a strategy named `StrategyName`.

State declarations may contain variable initializations. For example to initialize variable `commandline` with value `netscape` we use the following statement:

```
  add state StateName with strategy StrategyName::NS
    model {
```

```
    commandline = ‘‘netscape’’;
    };
```

In this example the strategy has a name space, `NS`, see Section 9.2.1.3 for a discussion of namespaces.

Internal and external transitions are declared separately. We can declare transition one at a time, indicating the source and the destination state as well as the label of the event triggering the transition, `from Source to Destination on Event;`.

The *chain declaration* of transitions is used to specify a sequence of transitions on the same event. For example instead of:

```
 {
  from S1 to S2 on success;
  from S2 to S3 on success;
  from S3 to Sfinal on success;}
 }
```

we can write

```
from S1 to S2 to S3 to Sfinal on success;
```

When transitions converge from multiple states to the same state, on the same event, instead of:

```
 {
 from S1 to ErrorHandler on failure;
 from S2 to ErrorHandler on failure;
 from S3 to ErrorHandler on failure;
 }
```

we can write

```
 on failure from S1,S2,S3 to ErrorHandler;
```

9.2.2.2 Initializing Model Variables. The blueprint can be used to initialize model variables. The model variable initialization is usually done after the agent description. This code is executed only once, when the agent is created. Blueprint recognizes three primitive variable types: strings, integers, and doubles. The initialization has a syntax similar with Java. For example we can write:

```
model {
  stringValue = ‘‘Hello world!’’;
  intValue = 1;
  doubleValue = 5.6;
}
```

We can also initialize the standard Java `Vector` and `Hashtable` types. The restriction is that the elements in both cases must be types accepted by blueprint (i.e.

strings, integers, floats, vectors or hash-tables). The keys of the hash-table must be strings.

The syntax is:

```
model {
 vectorValue = [1, 2.5, ``String''];
 hashtableValue = {First =``One'',Second=2,Third=3.0};
}
```

Complex structures can be created using multiply embedded vectors and hashtables:

```
model {
  complexStructure = { Name = ``Bond'',
           Type = ``AgentSystem'',
           Version = 2,
           Developers = [``boloni'', ``junkk'']
       }
}
```

We cannot initialize user-defined variables because their type may not be known to the agent factory.

9.2.2.3 Example. Now we present a simple agent which displays the "Hello Word" message, waits for user confirmation, then exits. The blueprint of this agent can be found in the `blueprint` directory in the Bond distribution:

```
 import bond.agent.strategydb;
 create agent HelloWorld
 plane Main
  add state Message with strategy Dialog.OkDialog
    model {
     Message="Hello, world!";
    };
  add state Exit with strategy Agent.Kill;

  internal transitions {
    from Message to Exit on success;
  }
 end plane;
 end create.
```

The first line `import bond.agent.strategydb;` specifies the path used by the agent to load its strategies. Then we describe the structure of a new agent called `Hello, World` with only one plane, `Main`. The state machine in that plane consists of:

(i) Two states one called `Message` with a strategy called `Dialog.OkDialog`, the other `Exit` with strategy `Agent.Kill`. The dot notation indicates that we are looking for a strategy called `OkDialog` from a *strategy group* called `Dialog`. This strategy displays a message box with a label and single button labeled `Ok`. The text of the label is read from the model, from a variable called `Message`. The strategy succeeds if the “Ok” button is pressed.

(ii) One internal transition between the two states.

The following commands start the agent editor and load the agent:
`RunAgent blueprint/HelloWorld.bpt` – on Linux
`java RunAgent blueprint/HelloWorld.bpt` – on Windows.
To start the agent directly:

`RunAgent -novisual blueprint/HelloWorld.bpt`.

In these examples we assumed that we are in the Bond directory, otherwise we have to specify the full paths.

9.2.3 Agent Transformations

A significant part of the inter-agent communication can be described as *control*: the behavior of the *controlled* agent is changed as a result of an action of a *controller* agent. The behavior of the agent is described by the state vector, and it can be changed by transitions, which alter one or more states of the state vector. One way to trigger transitions is by sending a message as shown in Figure 9.20.

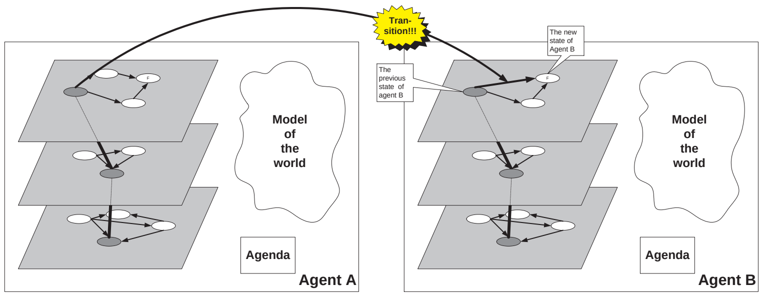


Fig. 9.20 Agent A desires to change the behavior of agent B, by changing a strategy on the first plane. It sends a message labeled with a transition name. The transition is performed on all planes of agent B where a match between an existing transition and the one in the message can be found. In this example we see only a match in the first plane.

Agents often cooperate for achieving certain goals. Cooperation requires knowledge sharing. In our structure this means that a segment of the model of one agent is copied to the model of another one. Information sharing is a very complex topic, we have to determine what part of the model will be shared, the identities of the agents, the confidence level in the shared knowledge and so on. Our system contains support for information sharing at the communication layer level, and contains various mechanisms to enforce security for inter-agent cooperation [24], [23]. Figure 9.21

presents an example of cooperation through knowledge sharing using the push mode. Agent A pushes part of its model to the model of agent B.

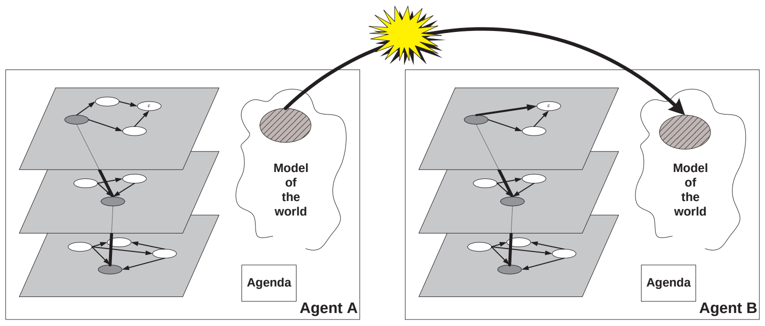


Fig. 9.21 Inter-agent cooperation using knowledge sharing. Agent A pushes part of its model into the model of agent B.

Joining and splitting are two useful operations facilitated by the multi-plane agent model. When *joining* two agents the new agent contains the planes of the two agents and the model of the resulting agent is created by merging the models of the two agents. We may separate the two models through the use of namespaces.

When *splitting* an agent we obtain two agents, the union of their planes gives us the planes of the original agent. The two agents need not be disjoint, some planes may be replicated. Both agents inherit the full model of the original agent.

There are several cases when joining or splitting agents are useful:

(i) Joining control agents from several sources, to provide a unified control,

(ii) Joining agents to reduce the memory footprint by eliminating replicated planes,

(iii) Joining agents to speed up communication,

(iv) Migrating only part of an agent,

(v) Splitting to apply different priorities to parts of the agent.

Another useful operation is *trimming*.The state machines describing the planes of an agent may contain states and transitions unreachable from the current state. These states may represent execution branches not chosen for the current run, or states already traversed and not to be entered again. The semantics of the agent does not allow some states to be entered again, e.g. the initialization code of an agent is entered only once.

Trimming is recommended to reduce the footprint of an agent before migration or checkpointing to limit the amount of data transferred in case of migration or stored in case of checkpointing or at runtime to reduce the memory footprint. Trimming is built into current agent migration code.

Determining the components to be trimmed is a problem in itself and requires reachability analysis. The Sethi-Ullman algorithm for reusing temporary variables from the theory of compiler construction [45] may be used to identify components that are no longer reachable.

9.2.4 Agent Extensions

Technologies for wide-area applications are continually evolving and an important design objective for any type of middleware is to be *open-ended*. Thus, a major concern in the design of the system described in this chapter is to integrate with ease new functions and to inter-operate with systems developed independently.

In this section we discuss three important extensions of the system. The objectives of these extensions are to: (i) improve the mobility of agents and their ability to communicate with one another and coordinate their actions; (ii) support fault detection and fault information dissemination in a federation of agents, and (iii) support inference.

So far we discussed only one aspect of agent mobility: the blueprint and the model are text files that can be transported with ease to a new location. Knowing the structure and the state of the agent, the agent factory at the new site may re-assemble and restart the agent. Yet, to be functional, the agent at the new site needs access to the strategies associated with the states of each plane of the agent. An agent may also need access to a blueprint to perform surgery, to adapt to changes in the new environment. Thus we need a societal service, a persistent storage server with a built-in access control mechanism where strategies and blueprints can be available for agents in need to share them.

Another problem is communication between strategies in the same plane, or in different planes of an agent, and, by extension communication between strategies of two different agents. The only mechanism available so far for strategies to communicate with one another was through the model of the agent, yet no methods supporting access control and concurrency control have been discussed yet. We had the choice of implementing a tuple space, a mailbox where items can be deposited and then retrieved from, or to integrate someone else's implementation.

The solution to both problems came in the form of a software developed at IBM Research called T Spaces [49]. The integration of tuplespace with the agent system is discussed in Section 9.2.4.1 and an application for synchronization of a group of Web monitoring and benchmarking agents is presented in Section 9.3.2.

Oftentimes agents have to work together to achieve a common goal. For example a federation of agents with different functions may be involved in monitoring and control of a Web server. The failure of any agent in the federation may either affect the functionality or the quality of the system. In Section 9.2.4.2 we discuss an extension to the system that allows agents in a federation to monitor each other and once a fault is detected to take corrective actions.

An orthogonal problem to mobility and fault tolerance is the intelligent agent behavior. As mentioned in Chapter 7, intelligence is necessary to guarantee autonomous behavior and has several dimensions: inference, learning, and planning. Inference provides agents with the ability to derive new facts, from a set of existing facts and a set of inference rules.

For example an agent may be dispatched to a new site and be required to install new software on that site. We do have the choice of a complex agent capable to work with any operating system, any hardware and software configuration or we may send a simple agent capable to discover basic facts about the site and then report them to a more sophisticated beneficiary that can use the facts to build a surgery blueprint to transform the original agent into a functional one. Again we had the choice to implement our own inference engine or to integrate an existing one. In Section 9.2.4.3 we discuss the integration of the Jess expert system shell [20] into our agent system and present an application to an adaptive MPEG server.

9.2.4.1 Extending the Model and Repositories for Blueprints and Stratgies: Tuplespaces. Tuplespaces extends message-passing systems with a simple persistent data repository that features associative addressing. They provide a powerful mechanism for interprocess communication and synchronization: a producer process generates a tuple and places it into the Tuplespace; a consumer process requests the tuple from the space. Tuplespaces have several distinctive features:

(i) Tuplespace communication is fully anonymous, the creator of a tuple does not need to have any knowledge about the future use of that tuple.

(ii) Tuplespaces allow time-disjoint processes to communicate seamlessly.

(iii) Tuplespaces use an associative addressing scheme and allow processes to communicate regardless of machine or platform boundaries.

The *Tuplespace* concept was originally proposed by Carriero and Gelernter as part of the Linda coordination language, [12], [13]. A Tuplespace is a globally shared, associatively addressed memory space that is organized as a bag of tuples.

A *tuple* is a vector of typed values, or fields. *Templates/anti-tuples,* are used to associatively address tuples via matching. A template is similar to a tuple, but some fields in the vector may be replaced by typed place holders called *formal fields*.

A formal field in a template is said to match a tuple field if they have the same type. If the template field is not formal, both fields must also have the same value. A template matches a tuple if they have an equal number of fields and each template field matches the corresponding tuple field.

The combination of Java and Tuplespace is pursued by projects such as Jada [16], JavaSpaces [54], and T Spaces, [28], [49] . Jada is a Linda implementation used to provide basic coordination for PageSpace [15] a high-level coordination system.

JavaSpaces, currently under development at Sun Microsystems, is designed to provide "distributed persistence" and aid in the implementation of distributed algorithms. The system allows arbitrary Java classes to be communicated as tuples and made persistent through Tuplespace. Transactions are provided for Tuplespace integrity, and a facility for notifying a process when a tuple is written to a Tuplespace is provided instead of the standard blocking read and take operations. JavaSpaces provides a simple transactional data repository and communication mechanism.

T Spaces is a software system developed in Java and available as freeware; it provides group communication services, database services, URL-based file transfer services, and event notification services.

The basic T Spaces tuple operations are `write`, `take`, and `read`. The `write` method stores its tuple argument in a Tuplespace. The `take` and `read` methods are non-blocking operations, each uses a tuple template argument that is matched

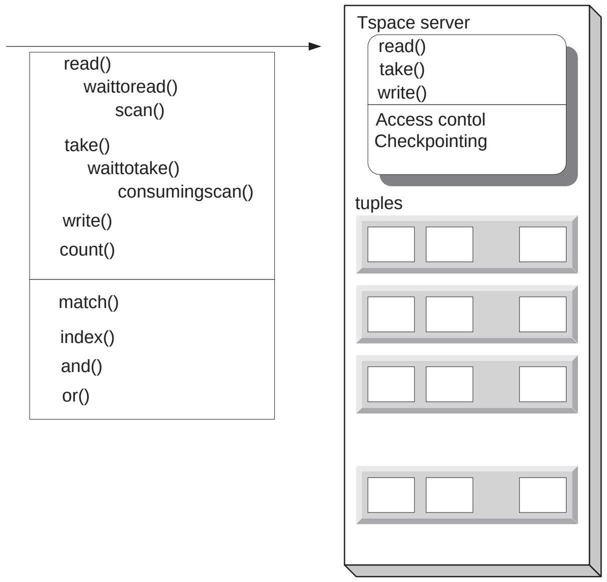


Fig. 9.22 A T Space server and the methods it supports.

against the tuples in a Tuplespace. Blocking versions of these, `waittotake` and `waittoread`, are supported; if no match is found these methods block until a matching tuple is written by another process.

The `take` method removes and returns the first matching tuple in the Tuplespace, whereas the `read` returns a copy of the matched tuple, leaving the Tuplespace unchanged. If no match is found, take and read each return the Java type null and leave the space unchanged. T Spaces also extends the standard Tuplespace API with the operations `scan`, `consumingscan`, and `count`. The `scan` and `consumingscan` methods are multiset versions of `read` and `take`, respectively, and return a "tuple of tuples" that matches the template argument. The `count` method returns an integer count of the matching tuples. Figure 9.22 shos a T Space server and the methods it supports.

In T Spaces a tuple matches the template when all of the following conditions hold:

(i) The tuple and template have the same number of fields.

(ii) Each of the fields of the tuple is an instance of the type of the corresponding field of the template.

(iii) For each non-formal field of the template, the value of the field matches the value of the corresponding tuple field.

T Spaces also provide several types of queries: `Match`, `Index`, `And`, and `Or` queries. A `Match` query performs structural or object compatibility matching, whereas an `Index` query performs a named-field query. `And` and `Or` queries can be used to combine these other queries and build complex query trees.

A T Spaces server is controlled by a configuration file, `tspaces.cfg` that specifies a wide range of parameters for the server such as:

- the port number the server listens to,

- a checkpoint file and the time interval between checkpointing the T Spaces server,

- time intervals to check for deadlocked threads and for expired tuples,

- access control parameters; if access checking is enabled, add/delete users or groups, access control lists.

Agent access to T Spaces. The `bondTupleSpaceEnabledStrategy` allows agents to communicate with one another via the T Space server. Moreover, strategies of different state machines of an agent can communicate with one another using tuplespaces provided that they extend the `bondTupleSpaceEnabledStrategy`.

This strategy extends the `bondDefaultStrategy`. Its `install()` action reads from the model the location of the T Space server and a string giving the tuple space name and sets up the tuple space.

```
 import com.ibm.tspaces.*;
 import bond.agent.*;
 import bond.agent.interfaces.*;
 import java.io.*;
 public class bondTupleSpaceEnabledStrategy extends
                               bondDefaultStrategy {
  protected TupleSpace space, save;
  boolean inited = false;

  public void install(bondFiniteStateMachine fsm) {
    super.install(fsm);
    if (!inited) {
      String host = (String)getModel("TupleServer");
      String sname = (String)getModel("SpaceName");
      if (host == null || sname == null) {
              inited = true; return;}
      inited = setTupleSpace(host, sname);
    }
  }
  public boolean setTupleSpace(String host, String sname) {
    try {
      space = new TupleSpace(sname, host);
      return true;
    }
    catch (TupleSpaceException e) {return false;}
    catch (Exception e) { return false;}
  }
}
```

The code for the actual blocking methods to take an item from the tuple space without leaving a copy, to read an item and leave the copy in the tuple space, and to write an item into the tuple spaces is shown below. These methods are wrappers for the methods supplied by the `com.ibm.tspaces` package: `waitToTake`, `waitToRead`, and `write`.

```
public Object getFromTupleSpace(String host,
        String sname, String s)throws Exception {
  if (!setTupleSpace(host, sname)) return null;
  return getFromTupleSpace(s);
}
public Object getFromTupleSpace(String s)
                               throws Exception {
  Tuple msg = space.waitToTake(s, new
                    Field(Serializable.class));
  return (Object)msg.getField(1).getValue();
}
public Object copyFromTupleSpace(String host,
       String sname, String s)throws Exception {
  if (!setTupleSpace(host, sname)) return null;
  return copyFromTupleSpace(s);
}
public Object copyFromTupleSpace(String s)
  throws Exception {
  Tuple msg = space.waitToRead(s, new
                   Field(Serializable.class));
  return (Object)msg.getField(1).getValue();
}
public boolean putIntoTupleSpace(String host, String
  sname, String s, Serializable o) throws Exception {
  if (!setTupleSpace(host, sname)) return false;
  return putIntoTupleSpace(s, o);
}
public boolean putIntoTupleSpace(String s,
                Serializable o)throws Exception{
    space.write(s, o);
    return true;
}
```

9.2.4.2 Fault Detection and Fault Information Dissemination.

We now provide details of the algorithm and the data structures.

Status table: it is a data structure containing fault–status information maintained by each agent. Let N be the total number of agents in a federation; some of them are faulty, others are fault–free. Consider an agent a, with aid_a, monitoring an agent b, with aid_b, and being monitored by c with aid_c. The status table maintained by agent a contains the following data:

A list of event–status counters for every other agent in the federation. $status[aid_i]$ is a non–negative integer value for the most recent "fail" or "join" event regarding agent i with aid_i. If $status[aid_i]$ is odd then agent i is faulty, if $status[aid_i]$ is even, agent i is fault-free.

The event–status counter provides information about the ordering of the events because it is incremented by one after each "fail" or "join" event regarding b, detected by a. When b joins the federation, and requests to be monitored by a, the counter is set to $status[aid_b] = 0$. When a detects a "fail" event of b, it increments the counter thus $status[aid_b] = 1$. When it detects a "join" event then it increments again, $status[aid_b] = 2$, and so on.

Recall that a monitors only one agent b and learns about failures detected by other agents through dissemination. In addition to "fail" events generated during monitoring, an agent may generate a "fail" event during the dissemination process when the contact agent fails to acknowledge a dissemination message.

A counter is only modified by an agent that has detected the occurrence of an event.

Monitoring: it keeps the AID of the agent that it is monitoring.

Monitored_By: it keeps the AID of the agent that is monitoring this agent.

The messages exchanged during the fault–detection and dissemination are:

`test-msg` and `fine` are a monitoring message sent by agent a to agent b it monitors, and a reply of b to a. Agent a expects the reply within a certain time interval. If the replay fails to materialize within that interval, a time-out occurs and a detects a "fail" event.

`info-msg` and `received` are a propagation message and an acknowledgment to the propagation. A propagation message contains: (i) the AID of the agent that generated the event, (ii) the value of the event status counter, (iii) the list of agent AID's the information should be forwarded to, and (iv) the list of the rest agents. The propagation continues until the forwarding list becomes empty. Unless the acknowledgment is received by the agent sending the message within a well–defined interval, a time-out occurs and it is considered as the "fail" event.

`request-monitoring`, `I-will-monitor-you` and `I-am-busy`. A new agent b sends a `request-monitoring` message to an agent a of the federation it knows about. Agent a may respond `I-am-busy` or `I-will-monitor-you`.

`request-join`, `I-will-monitor-you` and `I-am-busy` are a monitoring request message from a new or repaired agent and two possible replies: accept or deny. The reply messages also contain the list of fault-free agents to give the joining agent a hint about the current members of the federation.

`you-are-orphan` is a message to force a reconfiguration of the ring-monitoring topology when a new agent joins the federation. If agent a currently monitoring agent b receives a monitoring request from a new agent c and realizes that the ring topology forces it to accept to monitor c instead of b then it sends a `you-are-orphan` message to b. For example a has $aid_a = 10$, b has $aid_b = 20$. A new agent c with $aid_c = 15$ joins the federation and then the ring topology requires the new agent should be inserted between a and b and the monitoring relations be changed from $a \Rightarrow b$ to $a \Rightarrow c \Rightarrow b$.

The pseudo-code of the algorithm consists of a set of processes: *message_handler*, *info_handler*, *info_disseminator*, *monitor_searcher*. These processes run in parallel on separate execution threads and sometimes create instances of another processes.

Message Handler: it receives all the algorithm messages. Upon receiving a message, this process handles it or dispatches it to other processes.

```
process MESSAGE_HANDLER() {
while (TRUE) {
 receive message from agent i;
  switch (type of message)
   case TEST-MSG:
send FINE to agent i
   case INFO-MSG:
process__INFO_HANDLER(message, agent i)
   case REQUEST-MONITORING:
   if (procedure__Can_Monitor(agent i)) {
      process__FAULT_MONITOR(agent i)
      send I-WILL-MONITOR-YOU to agent i }
    else
      send I-AM-BUSY to agent i
   case REQUEST-JOIN:
    if (procedure__Can_Monitor(agent i)) {
      process__FAULT_MOINITOR(agent i)
      send I-WILL-MONITOR-YOU to agent i
      if (status[agent i] exists)
           status[agent i]++; /* set as fault free */
      else
           add status[agent i] = 0; /* add initialized one */
      process__INFO_DISSEMINATOR(agent i) }
    else
      send I-AM-BUSY to agent i
   case YOU-ARE-ORPHAN:
    set Monitored_By to null
    process__MONITOR_SEARCHER() }}
```

```
procedure boolean Can_Monitor(agent requester) {
if (Monitoring== null)
   return true;
cur_ID = the ID of the agent that it monitors
req_ID = the ID of agent requester
my_ID = the ID of this agent
if (my_ID < cur_ID)
  if (my_id < req_id && req_id < curr_id)
     return true /* accept request */
else if (my_DI > cur_ID) {
  if ((my_id > req_id && req_id < cur_id) || (req_id > my_id)) {
    return true /* accept request */
return false /* deny request */
```

normalsize

When the message_handler receives a request to monitor or join, it decides whether to accept or deny the request after checking its current monitoring state; if it does not monitor any agent, it accepts the request after verifying that the ring topology is satisfied. Otherwise it compares the AID of the requesting agent with its own and with that of the agent it currently monitors and then makes a decision subject to the condition that the ring topology is satisfied, see the procedure *Can_Monitor()*.

Fault Monitor: it monitors one agent by periodic polling. Once detecting a failure event, it starts a `info_disseminator` process to propagate the event to other fault–free agents.

```
process FAULT_MONITOR(agent i) {
  if (Monitoring != null) /*monitoring other agent */
    stop monitoring;
  Monitoring = agent i;   /* set new monitoring target */
  while (NOT STOPPED) {
    send TEST-MSG to agent i
    timed-wait FINE from agent i
    if (time-out)
      status[agent i]++; /* set as faulty */
      process__INFO_DISSEMINATOR(agent i)
      Monitoring = null;
      exit /* stop monitoring */
    wait for monitoring INTERVAL }
  if (STOPPED) {         /*forced to reconfigure */
    send YOU-ARE-ORPHAN to agent agent i
    timed-wait FINE from agent i
    if (time-out)
      process__INFO_DISSEMINATOR(agent i)
      status[agent i]++; }}/*set as faulty */
```

The time–out period for a reply message (*fine*) takes into account both message processing and network latency times. Once an agent detects a “fail” event, it increments its local status counter of the faulty agent by one to indicate a faulty agent.

Info Disseminator: it initiates the event dissemination. It constructs binary dissemination tree based on the snapshot of the current fault–free agents.

```
process INFO_DISSEMINATOR(agent i) {
  for all status[agent k] { /* collects all fault-free agents */
    if (status[agent k] ==even)
      Array fault-free[] += agent k }
  procedure__SPLIT_AND_SEND(agent i, fault-free)

procedure SPLIT_AND_SEND(agent event, list) {
  N = size of list[]
  Array list_1[] = list[0..N/2-1]  /* group 1*/
```

```
 3  agent x = random one of list_1[] /* contact agent of group 1 */
 4  Array list_2[] = list[N/2+1..N-1] /* group 2*/
 5  agent y = random one of list_2[] /* contact agent of group 2 */
 6  process__SPREAD_INFO(agent event, agent x, list_1)
 7  process__SPREAD_INFO(agent event, agent y, list_2)}

process SPREAD_INFO(agent event, agent receiver, list) {
 1 INFO-MSG = (event, list)
 2  send INFO-MSG to agent receiver
 3 timed-wait RECEIVED from agen receiver
 4  if (time-out) {
 5 status[agent receiver]++; /* set as faulty */
 6 process__INFO_DISSEMINATOR(agent receiver)
 7    if (list != null) {
 8      agent another_receiver = list[0];
 9      list = list[]  another_receiver;
 10     process__SPREAD_INFO(agent event, agent another_receiver, list)}}}
```

The procedure SPLIT_AND_SEND() splits the current list of fault–free agents into two groups and selects randomly two contact agents from each group. The time–out for the acknowledgment message (*received*) includes the time to tolerate the faults of the receiver agents during dissemination.

Info Handler: it handles the event messages propagated from other agents. After updating its local status table, it forwards the message to next level agents.

```
process INFO_HANDLER(message, agent sender) {
 1  if (more recent status[agent k] than local) {
 2    update local status[agent k]
 3    if (I am orphan)
 4      process__MONITOR_SEARCHER();
 5  }
 6  else if (older status[agent k]) {
 7    send RECEIVED to agent sender;
 8    return;}
 9  list[] = message.getList();
 10 if (list != null)
 11   procedure__SPLIT_AND_SEND(agent k, list)};
 12 if (the monitored agent is not in the dissemination list)
 13    forward the message to the monitored agent
 14 if (propagation ends)
 15    send acknowledgment
```

After updating the local status table of an agent, the info handler checks whether the value of $status[Monitored\_By]$ is odd, if so, it attempts to find another monitor. The acknowledgment is sent after the propagation to next–level contact agents is completed, to avoid the case thatleads to inconsistent status tables. In line 11 and 12, the agent checks whether the agent that it is monitoring receives the message, if not, it forwards the message to the monitored agent.

Monitor Searcher: it searches a fault–free agent which is able to monitor this agent. This process is initiated either when this agent joins a federation for first time, or when it finds itself an orphan.

```
process MONITOR_SEARCHER() {
   while (Monitored-By == null) {
  for all status[agent k] {
       if status[agent k] == even
          Array fault-free[] += agent k }
     agent target = procedure__CALCULATE_MONITOR(fault-free[]);
     if (a new joining or repaired agent) {
       send REQUEST-JOIN to agent target
       timed-wait reply from agent target
       if (time-out)
            continue /* try another agent */
        else {
          update local status table
          if (reply == I-WILL-MONITOR-YOU) {
            Monitored-By = agent target
            exit }
          else { /* I-AM-BUSY */
            continue }}}
     else { /* fault-free orphan agent */
       send REQUEST-MONITORING to agent target
       timed-wait reply from agent target
       if (time-out) {
           status[agent target]++; /* set as faulty */
           process__INFO_DISSEMINATOR(agent target)
          continue; }
       else {
          Monitored-By = agent target
          exit }}}}
```

```
procedure int CALCULATE_MONITOR(array agents[]) {
    N = size of agents[]
    sort(agents[]) /* sort agents[] in ascending order */
    /* get index of current agent */
    index i = binarySearch(my_agent_ID, agents[])
    if (i == 0)
       return the ID of agents[N-1]
       /* the largest ID agent should monitor */
    else
       return the ID of agents[i-1]}
```

The `CALCULATE_MONITOR()` procedure consists of the steps to obtain the AID of the agent to which a request message is sent to: sort the current list of fault–free agents in increasing order, find its position in the sorted list, select the AID of the agent preceding it.

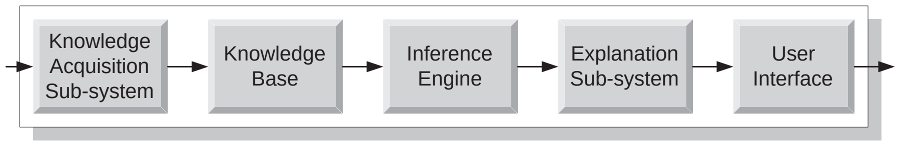


Fig. 9.23 The architecture of an expert system.

9.2.4.3 Integrating an Inference Engine. In this section we discuss the integration of an inference engine, Jess [20] in our agent system and in Section 9.3.1 we analyze in depth the application of inference for an adaptive video service.

The generic architecture of an expert system is presented in Figure 9.23. Its components are:

(i) Knowledge acquisition subsystem responsible to collect new facts.

(ii) Knowledge base, the store for factual and heuristic knowledge.

(iii) Inference engine, provides the inference mechanisms to manipulate symbolic information and knowledge.

(iv) An explanation system.

(v) A user interface.

An expert system shell consists only of an inference engine and a user interface.

Jess is a rule-based expert system shell written in Java. It applies continuously a set of `if-then` statements, the *rules* to a set of data, the *facts* in the knowledge base). An example of a rule, from Jess programming manual is:

```
(defrule library-rule-1
  (book (name ?X) (status late) (borrower ?Y))
  (borrower (name ?Y) (address ?Z))
 =>
  (send-late-notice ?X ?Y ?Z))
```

This rule says that if a book at a library is overdue a notification should be sent to the borrower. The facts here the name and status of the book and the name and address of the borrower.

A typical expert system has a fixed set of rules while the knowledge base changes continuously. The obvious implementation of an inference engine would be to keep a list of rules and continuously cycle through the list, checking each one's *left-hand-side, LHS* against the knowledge base and executing the *right-hand-side, RHS* of any rules that apply.

In the Rete algorithm, [20] the past test results are remembered across iterations of the rule loop. Only new facts are tested against any rule LHS. The computational complexity per iteration is linear in the size of the fact base.

Inference in Bond. All strategies using inference extend the `bondInferenceEngine` strategy. The code listed below shows the definition of the inference engine object and the execution of Jess commands.

```
import jess.*;
import bond.core.*;
import java.io.*;
public class bondInferenceEngine extends bondObject {
  public  Rete infegn;
  private StringBuffer kbase;
  public bondInferenceEngine() {
     infegn = new Rete();
     infegn.addUserpackage(new jess.ReflectFunctions());
     infegn.addUserpackage(new jess.StringFunctions());
  }
  public boolean executeCmd(String cmd) {
     try { infegn.executeCommand(cmd); return true;}
     catch (JessException e) {return false;}
  }
  public void run(int n) {
     try { infegn.run(n); }
     catch (JessException rexp) { }
  }
}
```

The Jess package provides a set of methods to add and retract facts to/from the knowledge base, `assertString(fact)` and `retractString(fact)`, to clear and reset the inference engine, `clear()` and `reset()`, to load facts into the knowledge base, `parse()`, to load rules, `append()`, to show the facts `ppFacts()`. Below we see the wrappers for these methods.

```
  public boolean insert_fact(String fact) {
     try {infegn.assertString(fact);}
     catch (JessException rexp) { return false;}
     return true;
  }
  public boolean remove_fact(String fact) {
     try {infegn.retractString(fact);}
     catch (JessException rexp) { return false;}
     return true;
  }
  public boolean clear_infegn() {
     try { infegn.clear();return true;}
     catch (JessException rexp) { return false;}
  }
  public boolean reset_infegn() {
     try { infegn.reset(); return true;}
     catch (JessException rexp) { return false;}
  }
  public boolean load_kbase(StringBuffer kbase) {
     if (kbase == null)  return false;
     this.kbase = kbase;
     StringReader sr = new StringReader(kbase.toString());
```

```
      Jesp jesp = new Jesp(sr, infegn);
       try {jesp.parse(false);}
      catch (JessException rexp) { return false;}
      return true;
   }
   public void loadRulefromFile(String fname) {
      try {
         RandomAccessFile f = new RandomAccessFile(fname, "r");
          StringBuffer s = new StringBuffer("");
          String str;
          while ( (str = f.readLine()) != null)
             s = s.append(str+"\n");
          if (!load_kbase(s)) {}
      }
      catch (FileNotFoundException e) {}
      catch (IOException e) {}
   }
   public boolean insertObject(String tmpltname, Object o) {
      try {
         Funcall f = new Funcall("definstance", this.infegn);
         f.add(new Value(tmpltname, RU.ATOM));
         f.add(new Value(o));
         f.execute(this.infegn.getGlobalContext());
      }
      catch (Exception e) { return false;}
      return true;
   }
   public String show_facts() {return infegn.ppFacts();}
```

9.3 APPLICATIONS OF THE FRAMEWORK

Now we discuss in depth two applications of the system presented in this Chapter and survey two others. The first example illustrates the design and implementation of an adaptive video service where a server agent uses an inference engine to select the data streaming mode based upon feedback from a client agent. Network congestion as well as limitations of the CPU cycles available at client and/or server sites are detected and stored as facts in a knowledge based. The server agent uses a set of rules to transmit a compressed video stream in the normal mode, to reserve communication bandwidth and/or CPU cycles at the client and server sites, or if reservations fail to drop frames or to transmit decoded frames if there is enough communication bandwidth but the client is unable to decode the frames at the desired rate.

The second application presents a Web monitoring and benchmarking service. A federation of monitoring agents install the benchmarking software on a set of client systems and then generate the requested workload. The monitoring agents work in lock step, the synchronization is provided by a tuplespace server.

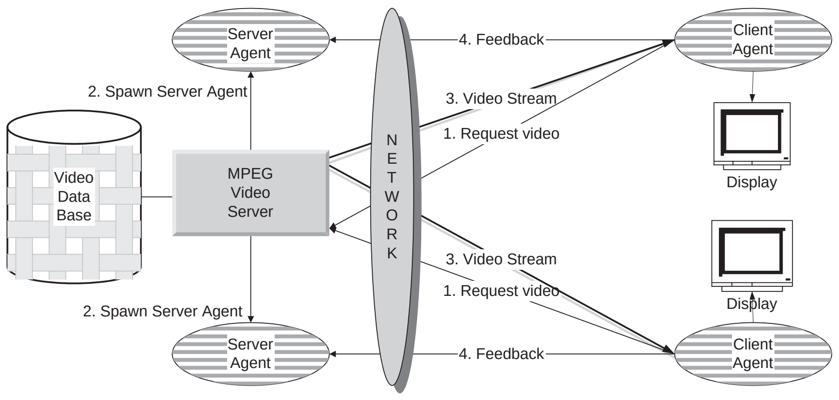


Fig. 9.24 The adaptive system consists of an MPEG server and server-client agent pairs supporting video streaming and display functions respectively. A server agent adapts to changing traffic and load conditions using a set of rules.

9.3.1 Adaptive Video Service

We now discuss an application of inference to support server reconfiguration and resource reservations for a video service, [32], [56].

The architecture of the adaptive MPEG system is shown in Figure 9.24. In response to a request from a client the MPEG video server spawns an MPEG server agent which delivers and controls the video streaming. The MPEG client agent displays the video stream, monitors its reception and provides feedback regarding desired and attained quality of service at the client side. The server agents responds by reconfiguring video streaming and reserving communication bandwidth and/or CPU cycles according to a set of rules. An inference engine, a component of the server agent, controls the adaptation mechanism. A native bandwidth scheduler and a CPU scheduler in Solaris 2.5.1 support QoS reservation as described in [56].

Two communication channels exist between a client and its peer on the server side: a channel for data streaming and a control channel for streaming commands and feedback from client to server as shown in Figure 9.24. We use UDP for video streaming with one frame per UDP packet. The packets arriving out of order are rearranged.

The *profile* of a video file is the data rate corresponding to a frame rate. Table 9.9 shows the profile of one of the video files we used for testing.

9.3.1.1 The Server Agent. A partial description of the blueprint for the server agent follows. The agent had two planes, one to for delivering the video stream and one to control the data streaming mode, see Figure 9.25. We only show the plane responsible for data streaming. As in other examples the error handling states are omitted. The two planes communicates with each other through the model.

Table 9.9 Profile of a sample application

Frame Rate (frames/sec).	Transmission Rate (bps))
5	4000
10	7000
15	10000
20	13000
25	16000
30	20000

```
 import bond.application.MPEG;
 create agent MPEGserver
 plane MPEGtransmit
  add state Init with strategy
    bondMPEGServerStrategyGroup.InitDataChannel;
  add state NormalMode with strategy
    bondMPEGServerStrategyGroup.TransVideoStream;
  add state DecodeMode with strategy
    bondMPEGServerStrategyGroup.TransPixelData;
  add state Drop&DecodeMode with strategy
    bondMPEGServerStrategyGroup.TransPixelDataWithDropping;

external transitions {
   from NormalMode to DecodeMode on gotoTransPixelData;
   from NormalNode to DropMode on gotoTransDroppedPixelData;
   from DecodeMode to NormalMode on
                 goFromTransPixelDataToTransVideoStream;
   from DropMode to NormalMode on
           goFromTransDroppedPixelDataToTransVideoStream;
}

 internal transitions {
   from Init to NormalMode on gotoTransVideoStream;
 }
 model {
   FILENAME = "bond/application/MPEG/Blazer.mpg";
   ALTFILENAME = "bond/application/MPEG/red.mpg";
 }
 end plane;
 plane MPEGcontrol
  .....
 end plane
 end create.
```

The server agent shown in in Figure 9.25 supports four streaming modes:

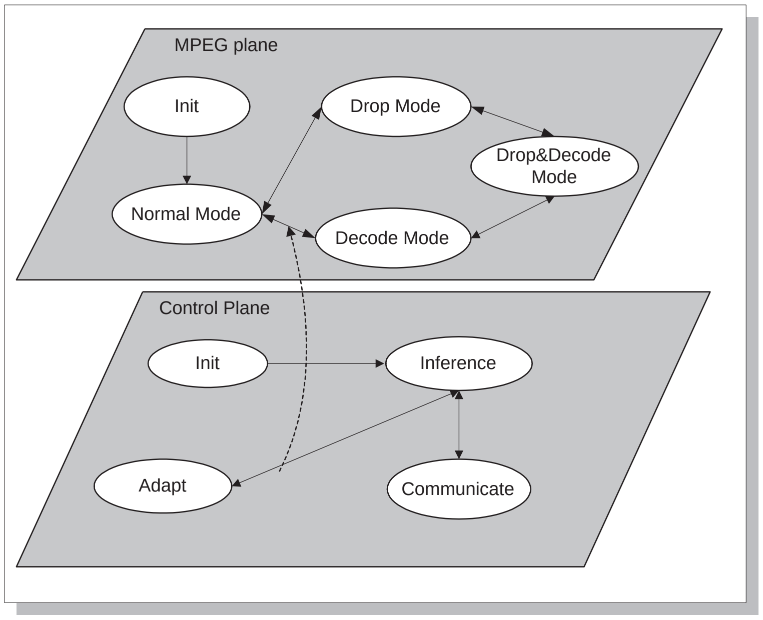


Fig. 9.25 The MPEG server agent has two planes.

Normal Mode The MPEG server reads the video stream from the Video Data Base or from a local file and transmits it to a client. The MPEG client decodes the video stream and displays the frames. Decoding the video stream is a CPU intensive operation.

Drop Mode. The MPEG server partially decodes the video stream to identify the frame types and drops certain type of frames. The sever selects the frames of which type affects the video quality less than other types, for example, B-type and P-type. This mode is suitable for low bandwidth.

Decode Mode. The MPEG server transmits decoded frames. Thus this mode is suitable for clients running on systems with CPU intensive programs, but connected via high-bandwidth networks.

Decode and Drop Mode. This mode is the combination of the previous two modes. It is suitable for overloaded clients connected with moderate bandwidth.

At the start of each transmission the server agent is in *normal mode* and as the network traffic and CPU load on the server and client change, the server agent reacts by selecting one of the other modes.

9.3.1.2 *The facts and the rules.*

The following facts are stored in the knowledge base:

Transmission rate in *bps* as measured by the server.

Packet loss rate. We detect lost packets by comparing the frame numbers on the client side. The packet loss rate is not the same as the frame loss rate, because P-frames and B-frames are dependent on I-frames. If an I-frame is lost, the depending frames are considered to be lost.

Inter-frame time. The inter-frame time shows the time elapsed at the client side between two displayed frames. I and P frames are larger than B frames, thus the number of operations and the time to decode them is larger.

Receiving rate. The client determines this rate using the information about packet sizes. This rate is affected by network congestion.

The rules for the resource reservation and reconfiguration are:
Bandwidth Reservation Rule. The objective of this rule is to reduce the packet loss rate by reserving bandwidth when the network is congested. The profile also has the maximum packet loss rate allowed to maintain a certain frame rate. We determine of the network is congested by comparing the `(packet-loss-rate)` with the `(maximum-loss-rate)` and, if so we reserve the bandwidth necessary to achieve the `(desired-frame-rate)`. The rule is:

```
(packet-loss-rate ?lr)
(desired-frame-rate ?fr)
(maximum-loss-rate ?mr)
(test (> ?lr ?mr))
=>
(reserve-bandwidth ?fr)
```

After this rule fires, the strategy of the *adapt* state in the control plane looks up the profile of the video transmission to determine the necessary bandwidth and passes the information to the bandwidth reservation interface.

CPU Reservation Rule. This rule fires when a CPU-intensive program running either at the server or the client affects the server transmission rate, or the inter-frame time at the client. The transmission rate of the server is compared with the profile and the inter-frame time is compared to the desired inter-frame time. This rule is repeatedly fired, and raises the reservation level gradually, until the desired rate is achieved. The rules are:

```
(transmit-rate ?tr)
(required-transmmit-rate ?rtr)
(test (< ?tr ?rtr))
=>
(increase-cpu-reservation)

(inter-frame-time ?ft)
(required-inter-frame-time ?rifr)
(test (< ?rifr ?ft))
=>
(increase-cpu-reservation)
```

Drop Rule. This rule fires when either the bandwidth or CPU reservation fails. In this rule new facts, `(bandwidth-reservation-failed)`, `(cpu-reservation-failed)` and are added to the knowledge base.

```
   (bandwidth-reservation-failed)
   =>
   (trigger-drop-mode)

   (cpu-reservation-failed)
   =>
   (trigger-drop-mode)
```

We now present the actual facts and rules used by the server agent. At first, we see the definition of different rates measured by the server or reported by its peer client. Then there are several rules to maintain the facts in the knowledge base. Each measurement carries a time stamp and the fact corresponding to an older measurement is retracted. We only show one of these maintenannce rules.

```
;; Target frame rate the server wants to reach
 (deftemplate current-server-frame-rate
   (slot timestamp)(slot rate))
;; Actual frame rate measured
 (deftemplate actual-server-frame-rate
   (slot timestamp)(slot rate))
;; Server transmission rate
 (deftemplate transmit-rate-bytes-per-sec
   (slot timestamp)(slot interval) (slot rate))
;; Current mode of operation
 (deftemplate tcurrentmode
   (slot stime) (slot mode))
;; Client receiving rate
 (deftemplate receiving-rate-bytes-per-sec
   (slot timestamp)(slot interval)(slot rate))
;; Frame loss rate reported by client
 (deftemplate frame-loss-rate-frames-per-sec
   (slot timestamp) (slot interval) (slot rate))
;; Display rate reported by client
 (deftemplate display-rate-frames-per-sec
   (slot timestamp)(slot interval)(slot rate))

(assert (minimum-frame-rate 20.0))

;; Server actual frame rate maintenance rule
 (defrule actual-rate-maintenance
   (declare (salience 100))
   ?ar1 <- (actual-server-frame-rate (timestamp ?ts1))
   ?ar2 <- (actual-server-frame-rate (timestamp ?ts2))
   (test (< ?ts1 ?ts2))
   =>
```

```
    (retract ?ar1))
  ;; Server transmission rate maintenance rule
   .........
  ;; Current mode maintenance rule
   .........
  ;; Receiving rate maintenance rule
   .........
  ;; Frame loss rate maintenance rule
   .........
  ;; Display rate maintenance rule
   .........
  ;; Server rate maintenance rule
   .........

  ;; Decrease server frame rate rule
  (defrule decrease-server-frame-rate
    (tcurrentmode (mode normal))
    (current-server-frame-rate (rate ?r))
    (actual-server-frame-rate (rate ?r1))
    (display-rate-frames-per-sec (rate ?r2))
    (test (< (/ ?r2 ?r1)(/ 90 100)))
    =>
    (printout t "Decrease server frame rate to
        " (- ?r 1) ": current display rate--> " ?r 2 crlf)
    (if (< 2 ?r) then
     (call (fetch MODEL) setModelFloat "frameRate"(- ?r 1))
    else
     (call (fetch MODEL) setModelFloat "frameRate" 1.0)))
  ;; Increase server frame rate rule
  (defrule increase-server-frame-rate
    (tcurrentmode (mode normal))
    (current-server-frame-rate (rate ?r))
    (actual-server-frame-rate (rate ?r1))
    (display-rate-frames-per-sec (rate ?r2))
    (minimum-frame-rate ?mr)
    (test (< ?r2 ?mr))
    (test (>= (/ ?r2 ?r1) (/ 90 100)))
    =>
    (printout t "Increase frame rate to " (+ ?r 2)  "
                 : current display rate--> " ?r2 crlf)
    (call (fetch MODEL) setModelFloat "frameRate" (+ ?r 2)))
;; Measurement rule
  (defrule measurement(tcurrentmode (mode normal))
    (actual-server-frame-rate (timestamp ?ar))
    (display-rate-frames-per-sec (rate ?r) (timestamp ?t))
    (minimum-frame-rate ?mr)
    (test (<= ?r ?mr))
    (not (under-minimum-frame-rate))
    =>
    (assert (under-minimum-frame-rate)) ; flag
    (assert (under-minimum-frame-rate-since ?ar)))
 ;; Reset mode rule
  (defrule reset
    (tcurrentmode (mode normal))
    (display-rate-frames-per-sec (rate ?r) (timestamp ?t))
    (minimum-frame-rate ?mr)
    (test (> ?r ?mr))
    ?x<-(under-minimum-frame-rate)
    ?y<-(under-minimum-frame-rate-since ?k)
    =>
    (retract ?x)
    (retract ?y))
 ;; Test if under minimum frame rate rule
  (defrule under-minimum-rate
    (tcurrentmode (mode normal))
    (display-rate-frames-per-sec (rate ?r) (timestamp ?t))
    (minimum-frame-rate ?mr)
    (test (<= ?r ?mr))
    ?x<-(under-minimum-frame-rate)
    ?y<-(under-minimum-frame-rate-since ?ts)
    (test (> (- (* (time) 1000) ?ts) 30000)) ;
             under-minimum-rate continues over 30 sec.
    =>
    (printout t "**To Drop>>" crlf)
    (retract ?x)
    (retract ?y)
    (call (new bond.application.MPEG.MPEGAdaptation)
                adapt Normal Drop (fetch MODEL)))
 ;; Go back to normal rule
  (defrule go-back-to-normal
    (tcurrentmode (mode drop) (stime ?st))
    (actual-server-frame-rate (timestamp ?ts))
    (test (> (- ?ts ?st) 30000)) ; retry after 30 sec.
    =>
    (printout t "**To Normal>>" crlf)
    (call (new bond.application.MPEG.MPEGAdaptation)
                 adapt Drop Normal (fetch MODEL)))


  (deffunction
   fetch-from-model (?a) ;;   (call (fetch MODEL) getModel ?a))
```

9.3.1.3 The strategies. The `bondMPEGStrategyGroup` object provides the strategies associated with the states of the server agent. Here we only show the strategies for the *init* and *normal mode* states in the *MPEG plane*. The intializatation strategy identifies the thread handling a new connection to the video server and writes this information in the model. Then it causes a transition to the *normal mode* state. The strategy associated with the normal transmission mode in its `install()` function first reads from the model the name of the video file to be transmitted and the name of the coordinator, then initiates the transmission of the UDP stream, and finally writes into the model the name of the current state.

```
public bondMPEGServerStrategyGroup(String name) {
super(name);
// 1. The strategy for the INIT state of the server
strat = new bondDefaultStrategy() {
  OutputStream os = null;
  boolean errorFlag = false;
  public void install(bondFiniteStateMachine fsm) {
     super.install(fsm); }
  public long action(bondModel m, bondAgenda a) {
     setModel("MPEGServerThreadGroup",
        new ThreadGroup("MPEGServerThreadGroup"));
     if (!errorFlag)
        transition("gotoTransVideoStream");
     else
        transition("gotoError"); return 1000L;}
};
addStrategy("InitDataChannel", strat);

// 2. The strategy for the Normal Mode
strat = new bondDefaultStrategy() {
  UDPTransmitter ut;
  public void install(bondFiniteStateMachine fsm) {
    super.install(fsm);
    ut = new UDPTransmitter(this, (String)getModel("FILENAME"));
    setModel("UDPTransmitter", ut);
    bondCoordinator bc = (bondCoordinator)getModel("COORDINATOR");
    bc.insert_fact("(tcurrentmode (mode normal)
          (stime "+System.currentTimeMillis()+" ))");
  }
  public long action(bondModel m, bondAgenda a) {
    return 10000L; }
  public void uninstall() {
    ut.stop();
    dir.unregister(ut);
    ut = null;
  }
  public String getDescription() {
               return "Transmit video stream";}
};
addStrategy("TransVideoStream", strat);
```

9.3.1.4 The client agent. The `run` method of the `MpegDisplay` object used to display data on the client side is listed below. It reads from the input the data stream and uses the `display` function of a Java MPEG player to display a frame with a given sequence number and a given type. Periodically, it sends to the server agent a report with a time stamp, the interval between two consecutive measurements, and the display rate.

```
public void run() {
  String mpegserver =(String)bds.getModel("mpegserver");
  int port = ((Integer)bds.getModel("portnumber")).intValue();
  try {
   Socket s = createSocket(mpegserver, port, port);
   ois = new ObjectInputStream(new BufferedInputStream
                                         (s.getInputStream()));
  }
  catch (StreamCorruptedException sce) { }
  catch (IOException ioe) { }
  mpd = new MPEG_Play_Decoding((JFrame)bds.
                                    getModel("DisplayFrame"));
  while (!finish) {
   try {
    ap = (AnotherPacket)ois.readObject();
    if (first) {
       Runnable r = new Runnable() {
         public void run() {
         mpd.set_dim(ap.width, ap.height, ap.ori_w, ap.ori_h);}
       };
       SwingUtilities.invokeAndWait(r);
       first = false;
       mpd.display(ap.picture, ap.num, ap.type);
       long t1 = System.currentTimeMillis();
        if ((t1-lastDisplayMeasureTime) >
                               DisplayRateMeasureInterval) {
         double rate = ((num_of_frames+1)*1000)
                                /(t1-lastDisplayMeasureTime);
         fm.sendFeedback("(display-rate-frames-per-sec "+
                          "(timestamp "+t1+" )"+"(interval+"
          DisplayRateMeasureInterval+")"+"(rate "+rate+"))");
         num_of_frames = 0;
         lastDisplayMeasureTime = t1;
        }
        else {num_of_frames++;}
        catch (EOFException eofe) {}
        catch (IOException ioe) { }
        catch (ClassNotFoundException cnfe) { }
        catch (Exception e) {}
   }
  }
 }
```

The client agent measures the display rate, the frame loss rate, and the actual data rate and provides feedback to its peer server agent that changes its streaming mode accordingly. Our adaptation strategy is based upon the observation that the bottleneck can be any of the three resources along the end-to-end streaming path, the server CPU, the network, and/or the client CPU. The system identifies the bottleneck as follows: if the server transmission rate is below a minimum rate, then the CPU on the server side is a bottleneck; if the packet loss rate measured as the difference between the sender frame rate and the receiver frame rate, is high, then the network is the bottleneck; if the inter-frame display time at the client exceeds a threshold and the network is not congested, then the CPU on the client side is the bottleneck.

9.3.1.5 Experiments. In this section we present measurements for the MPEG application with and without resource reservation. In this experiment the server runs on an Ultra Sparc-1 machine with 128 MBytes memory, under Solaris 2.5.1. The client runs on a Pentium II 300 MHz, with 128 MBytes memory system under Solaris 2.5.1. To simulate increased traffic load a communication-intensive program generates a burst of UDP packets. To simulate the CPU load, CPU intensive program is used.

The first experiment shows the effect of bandwidth reservation, see Figure 9.26. On the server side in addition to the MPEG application we start a communication-intensive program. The traffic generated by this application affects the video traffic and we study the effect of this interference. Without reservation a large percentage of video packets are lost. Once sufficient bandwidth to support the desired frame rate is reserved, the number of lost packets is noticeably reduced, even under the heavy network traffic.

The second experiment shows the effect of CPU reservations, see Figure 9.27. We run three CPU-intensive processes to compete for with the MPEG process. The experiment is performed first without CPU reservation, and then with reservation. shows the effect of the CPU reservation on the inter-frame time. We start the CPU-intensive processes while the client displays frame 120 and then stop it around frame 230.

9.3.2 Web Server Monitoring and Benchmarking

9.3.2.1 Introduction. In this section we present a monitoring and benchmarking system as an example of coordination of Web-based activities. The system is described in detail in [30] and [34].

The widespread use of web servers for business-oriented activities requires some form of quality of service guarantees; short-term unavailability of services and large variations of the response time may have a severe negative economic impact. Yet, providing quality of service guarantees is a complex problem with multiple facets. One of the them is the ability to continually monitor a Web server and subject it periodically to realistic benchmarks.

Web monitoring and benchmarking require several entities distributed over a wide-area network to work together. Given that the number of clients of a Web server is very large, multiple client machines are often required to generate a realistic workload.

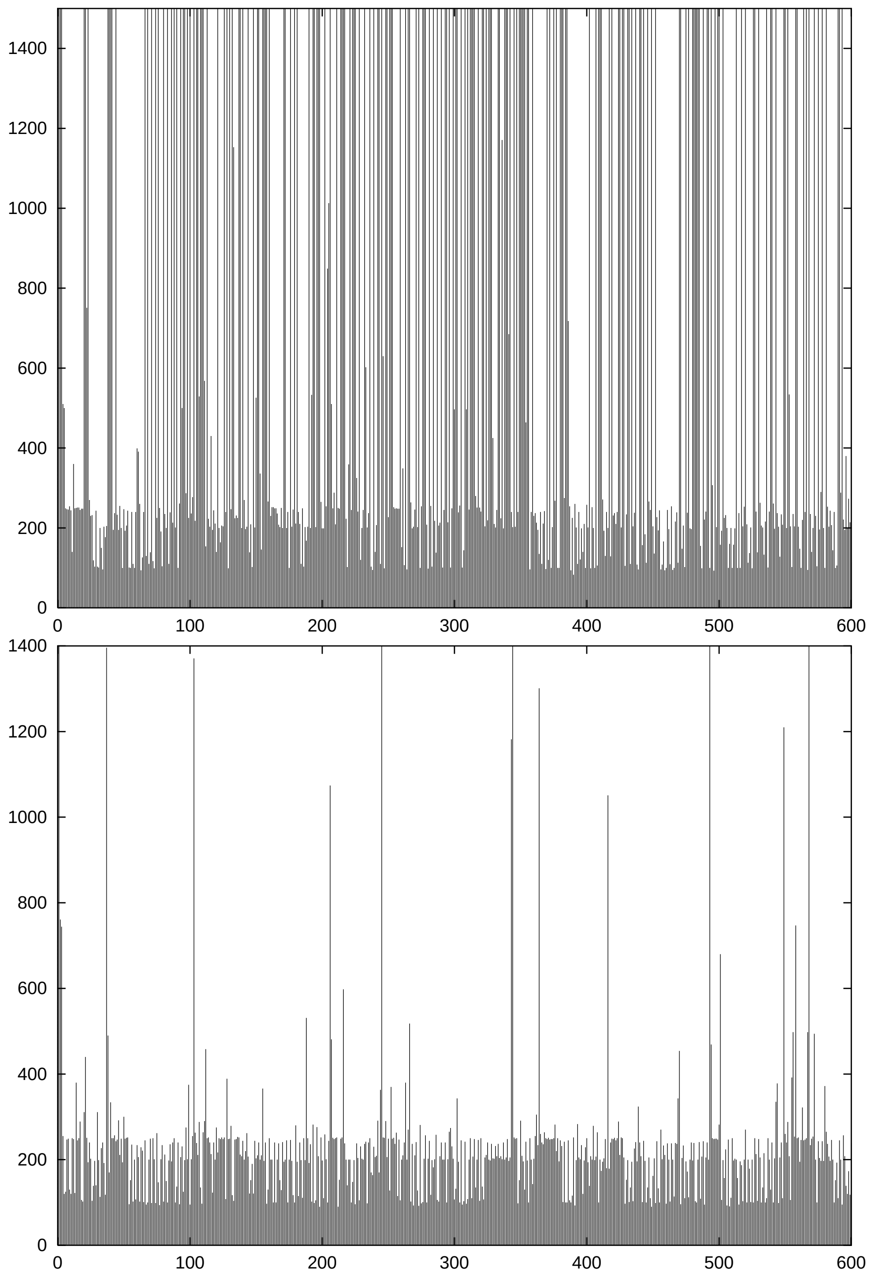


Fig. 9.26 The effect of the bandwidth reservation. The graph shows the inter-frame times measured at the client site, with the frame number on the horizontal axis and time in milliseconds on the vertical axis. The inter-frame time for lost frames is set to the maximum value, 60000 milliseconds, thus lost frames appear as vertical lines in the graph. Without reservation a large percentage of video packets are lost, as shown in the upper graph. Once sufficient bandwidth to support the desired frame rate is reserved, the number of lost packets is noticeably reduced, even under the heavy network traffic, see lower graph.

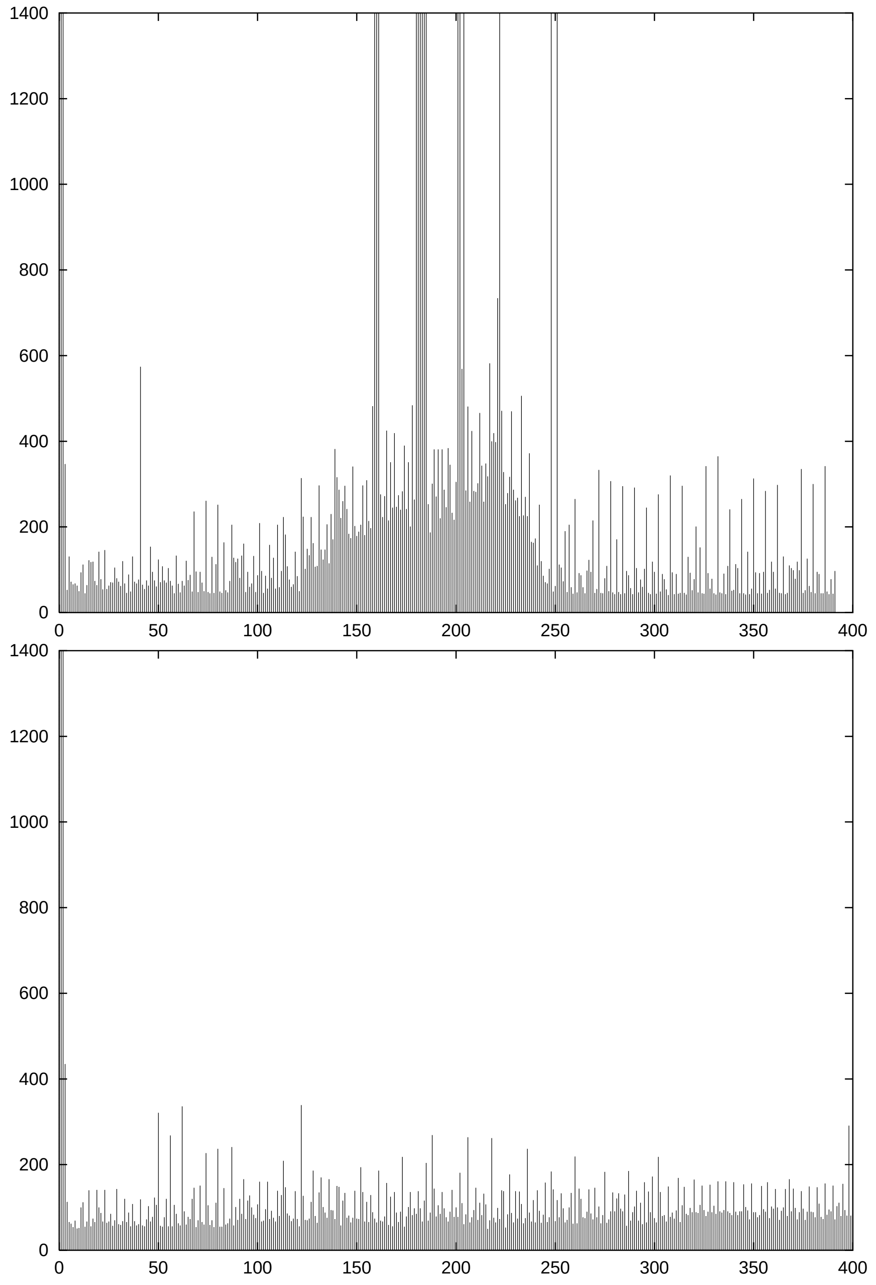


Fig. 9.27 The effect of CPU reservation. The graphs show the inter-frame time for individual frames at the client side when the MPEG display process competes with another CPU intensive process. In the upper graph, the CPU-intensive process is started while the client is running and then stopped. In the lower graph, the CPU-intensive program does not affect the client's ability to process the MPEG frames due to CPU reservation. The frame numbers appear on the horizontal axis.

Multiple monitoring points scattered throughout the network are necessary to simulate user actions.

Several commercial web monitoring service companies exist today, [38], [26]. Generally they provide a static service, the locations of the clients and the workload they generate are fixed and rarely emulate the behavior of real-life clients. The next generation web monitoring services are expected to address the problem of client mobility and of accuracy of benchmarking.

The functionality of existing benchmark suites and monitoring tools can be extended using mobile agent technology. Mobile agents have several advantages over the existing techniques:

Software installation: once a mobile agent is deployed at a site it can download the software tools for benchmarking and measurements, compile, and install them without human intervention.

Complex tasks: the mobile agents supervising data collection and analysis can perform their task autonomously and assist in performing complex measurements and data analysis tasks that require inference and/or planning.

Coordination: the agents can coordinate the measurements performed by multiple tools. They can provide coordination primitives for data collection and analysis, such as barrier–synchronization [30] and event notification.

Efficient data analysis: large volume of data can be processed by dispatching mobile agents to the data site rather than moving the data. In addition, the mobile agents can migrate among network nodes to process the measurement data, which are distributed over a set of client machines.

This section is organized as follows: first we survey a tool capable to generate synthetic workloads, then we describe the architecture of our system and discuss its advantages.

9.3.2.2 Surge - a Workload Generator for Web Servers. Several tools to generate synthetic workloads are available:

HTTPerf [41] uses multiple processes to generate HTTP requests at a fixed rate, a situation rarely encountered in real–world.

SpecWeb [46] is a Web benchmark software developed by a of industry and university researchers. It measures the maximum simultaneous number of connections that a Web server can sustain.

WebStone [40] and WebBench [55] provide similar benchmark softwares and directions.

TPC Benchmark W (TPC-W) [50] is a benchmark specification to test the transactional functionality of Web servers for electronic commerce.

Surge [1] is a software system which generates realistic Web workloads based upon six empirical statistics: server file size distribution, request size distribution, relative file popularity, embedded file references, temporal locality of reference, and user think times.

The architecture of this tool separates the problem of creating the workload from the methodology for benchmarking. Surge consists of three components: *workload data generator*, *client request generator*, and *server file generator*.

The workload data generator creates workload datasets which specify the file size distribution, the request sequence, the number of embedded files in each requested file, and the sequence of user think times. Both the server file generator and the client request generator rely on the generated datasets to perform their tasks. The server file generator creates a set of files matching the file size distribution of the dataset. The files are placed into a document subtree of a tested Web server. The client request generator, a multi–threaded process, each thread simulating one user, makes HTTP requests as specified in the dataset. Multiple client request generators on different machines can be used in one benchmark.

9.3.2.3 Web Benchmarking Agents. The Web benchmarking procedure consists of four steps: software installation, workload dataset generation, request generation, and analysis of measurements. At each step multiple*monitoring agents* perform the tasks required by that step. The control agent on the server site installs the software system to generate the files used in the benchmarking process and then activates the client. The monitoring and the control agent are supervised by a *coordinator agent*.

The flow of control in the benchmarking process is described in Figure 9.28. The benchmarking process is initiated when the beneficiary, in this case the individual conducting the benchmarking experiment, uses the visual interface to send an `assemble-agent` message to the agent factory running on the system hosting the coordinator. This message contains the address of the blueprint repository and the path of the blueprint for the coordinator agent. This blueprint is presented later in this section.

When the agent factory uses the blueprint to assemble the coordinator agent. When the agent assembly is completed, the agent factory sends an `agent-created` message to the beneficiary and provides the *bondID* of the agent. Next, the beneficiary sends a `start-agent` message for the agent identified by the *bondID* to the resident at `coordinatorIPaddress:2000`. This message includes the model of the agent.

The model is an XML description of the information needed by the coordinator to create and control monitoring agents for each Web client as well as the control agent for the Web server. The model description consists of six vectors, four for the monitoring agents on the client side and two for the control agent on the server side. Each client vector is named after the corresponding benchmarking step and consists of three hashtables, one for each client. Each hashtable provides a pair name and value for three strings identifying the host, the path on that host to the blueprint for the monitoring agent, and the path to the model. There are two vectors for the control agent on the server side. The agent has to install the file generation software and then to activate it.

```
<?xml version="1.0"?> <Model>
 <! Install SURGE on Web clients >
 <Vector name="SoftwareInst.Agents">
```

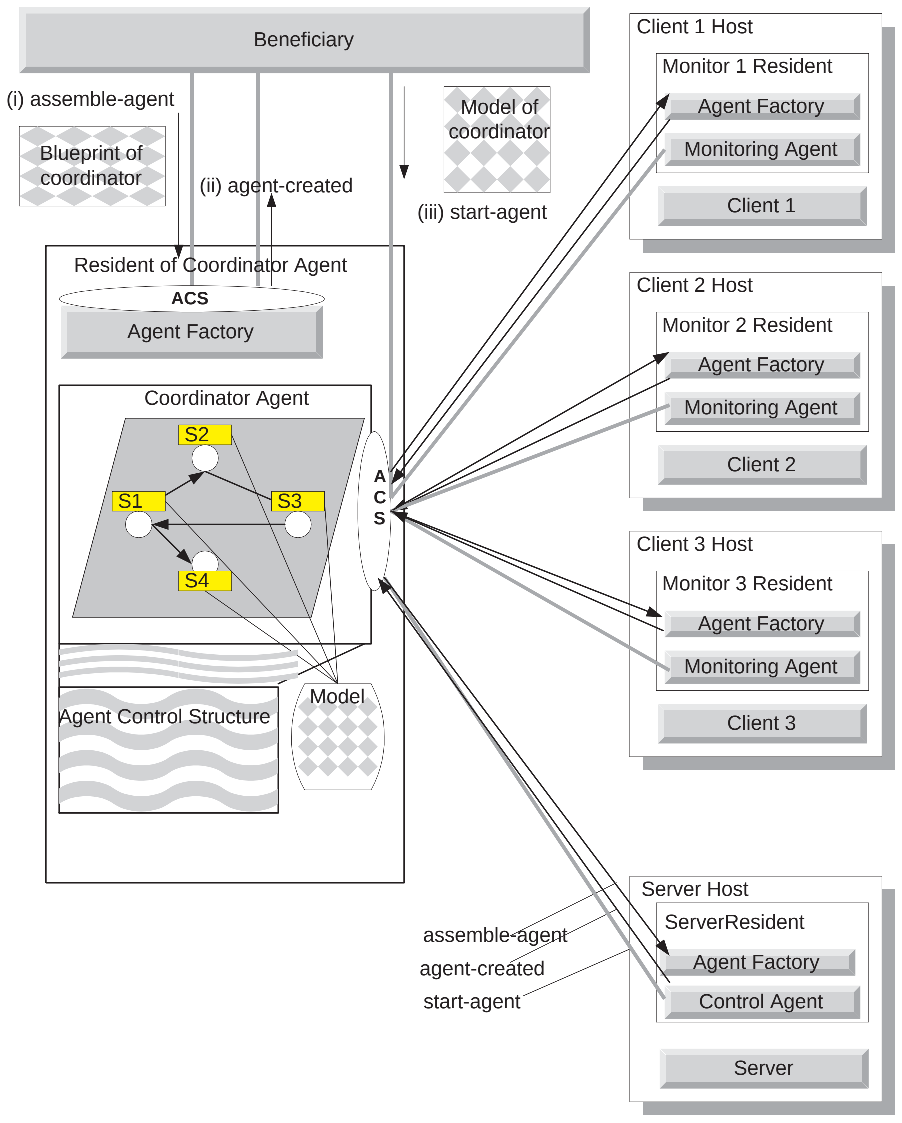


Fig. 9.28 Agent–based Web benchmarking system. The beneficiary triggers the assembly and the startup of the coordinator agent. The blueprint and the model of the coordinator agent are supplied by the beneficiary in the `assemble-agent` and `startup-agent` messages. Once started, the coordinator uses information in its model to start-up the three monitoring agents on sites where the clients are located as well as the control agent on the Web server site.

```
 <Hashtable>
 <String name="RemoteAddress">c1Host:2000</String>
 <String name="Blueprint">c1Bpt/SoftInst.bpt</String>
 <String name="Model">c1Mod/workloadclient.xml</String>
 </Hashtable>
  ...........
 </Vector>

<! Generate workload data; client-side command execution>
 <Vector name="WorkloadGenCmdExec.Agents">
 <Hashtable>
 <String name="RemoteAddress">c1Host:2000</String>
 <String name="Blueprint">c1Bpt/WorkloadGenCmdExec.bpt</String>
 <String name="Model">c1Model/WorkloadGenCmdExec.xml</String>
 </Hashtable>
    .........
 </Vector>

<! SURGE workload generation>
 <Vector name="StartGeneration.Agents">
 <Hashtable>
 <String name="RemoteAddress">c1Host:2000</String>
    <String name="Blueprint">c1Bpt/WorkloadGen.bpt</String>
    <String name="Model">c1Mod/WorkloadGen.xml</String>
    ......
  </Hashtable>
  </Vector>

<! SURGE data analysis>
 <Vector name="Analysis.Agents">
 <Hashtable>
 <String name="RemoteAddress">c1Host:2000</String>
    <String name="Blueprint">c1Bpt/loganal.bpt</String>
    <String name="Model">c1Mod/loganal.xml</String>
    ......
  </Hashtable>
  </Vector>

 <! Install file generator software on Web server>
 <Vector name="WebFileSoftInst.Agents">
 <Hashtable>
 <String name="RemoteAddress">sHost:2000</String>
 <String name="Blueprint">sBlpt/SoftInst.bpt</String>
 <String name="Model">sModel/workloadfile.xml</String>
 </Hashtable>
 </Vector>
 <Integer name="WebFileSoftInst.Interval">1000</Integer>

<! Server-side Command execution>
```

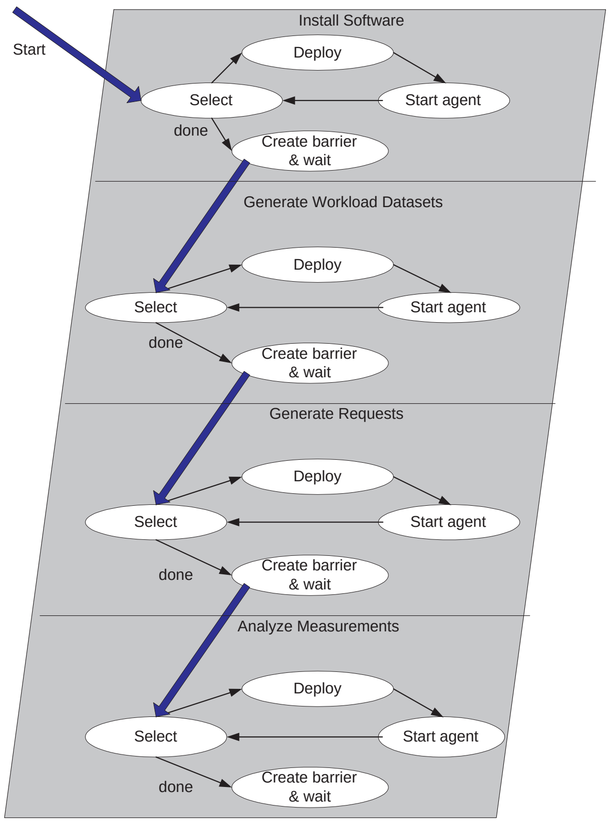


Fig. 9.29 A representation of the blueprint of the coordinator agent restricted to the client-side agents. It shows the states and transitions among them, where the states are grouped to form workflows corresponding to the four steps of the Web benchmark.

```
 <Vector name="WorkloadGenCmdExecServer.Agents">
 <Hashtable>
 <String name="RemoteAddress">sHost:2000</String>
 <String name="Blueprint">sBpt/WorkloadGenCmdExec.bpt</String>
 <String name="Model">sMod/WorkloadGenCmdExecServer.xml</String>
 </Hashtable>
 </Vector>
</Model>
```

The blueprint for the coordination agent consists of one plane only. In this plane there are four groups of states each corresponding to one benchmarking step at the client site and two groups corresponding to the software installation and activation at the server site. We only discuss the client section of the blueprint for the coordinator agent. Recall that a monitoring agent goes through the following steps:

(i) Install the Surge tools on the client machine.

(ii) Use the Surge tools to generate the workload description files to be used by the client processes.

(iii) Start-up client processes that generate the HTTP requests.

(iv) Start-up the data analysis tools.

For each of the four steps described above the coordinator uses the information provided by its model to create and control each monitoring agent. The coordinator model gives: (a) the host; (b) the path to the blueprint, and (b) the path to the model for each agent. Figure 9.29 shows that the coordinator goes through the following states in each step:

1. Locate the hosts where the agents are expected to run.
2. Request the respective agent factories to assemble each agent using the blueprint.
3. Start up each agent using the information in the model
4. Wait until all agents in the group have completed their execution.

The `bondMultiAgentDeployStrategy` uses different namespaces:
`::SoftwareInst` to install Surge and
`::StartGeneration` for measurements.

A partial blueprint for the coordinator agent consisting of only two steps, installation of the Surge and generation of HTTP requests follows.

```
import bond.agent.strategydb;
importbond.agent.strategydb.barrier;
create agent WebCoordinationAgent
plane Control
// instal Surge
add state SoftwareInst with strategy
     bondMultiAgentDeployStrategy::SoftwareInst;
add state Deploy with strategy
    bondAgentExecStrategyGroup.CreateAgent;
add state Start with strategy
          bondAgentExecStrategyGroup.StartAgent;
add state FirstBarrier with strategy
    bondBarrierWaitStrategy::FirstBarrier;
// measurements
add state StartGeneration with strategy
     bondMultiAgentDeployStrategy::StartGeneration;
add state DeployForGeneration with strategy
            bondAgentExecStrategyGroup.CreateAgent;
add state StartForGeneration with strategy
             bondAgentExecStrategyGroup.StartAgent;
add state GenerationBarrier with strategy
        bondBarrierWaitStrategy::GenerationBarrier;

internal transitions {
// Install SURGE
   from SoftwareInst to Deploy on success;
```

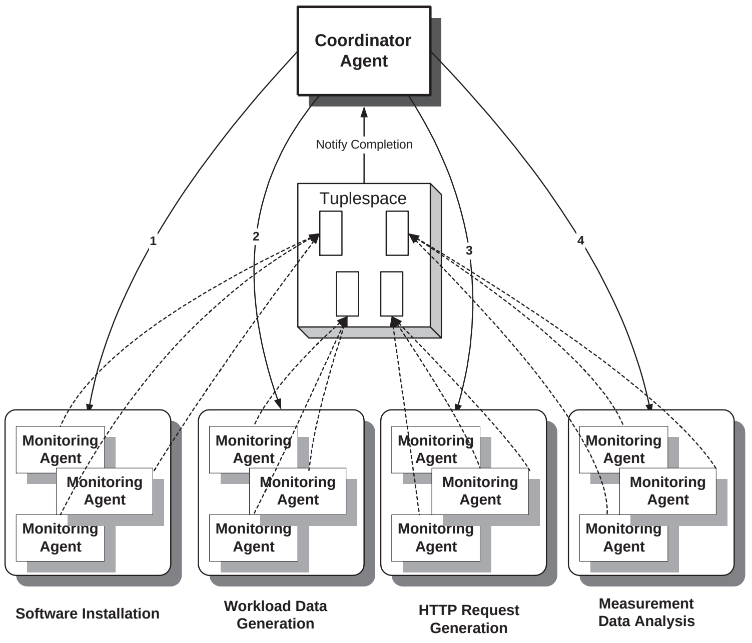


Fig. 9.30 Coordination of monitoring agents in the agent–based Web benchmarking system. The coordinator agent supervises a group of three monitoring agents and leads them through each step. All monitoring agents in the group have to complete a step before the next one is initiated. The barrier-synchronization is implemented by a tuplespace.

```
   from Deploy to Start on success;
   from Start to SoftwareInst on success;
   from SoftwareInst to FirstBarrier on next;
   from FirstBarrier to WebFileSoftInst on success;
// Generate HTTP requests
   from StartGeneration to DeployForGeneration on success;
   from DeployForGeneration to StartForGeneration on success;
   from StartForGeneration to StartGeneration on success;
   from StartGeneration to GenerationBarrier on next;
   from GenerationBarrier to GenerationSuccess on success;
}
end plane;
end create.
```

When the coordinator agent initiates a step, it creates a barrier in the tuplespace then starts up a set of monitoring agents that have identical tasks. Each monitoring agent is assigned the tasks required for that step. After finishing a task a monitoring agent deposits a token in the tuplespace. When the specified number of tokens was collected, the control agent is notified and it proceeds to the next step.

During the software installation step, a monitoring agent downloads from a Web server the Surge tools, currently the C language version, and then compiles them using the C compiler and the libraries. Even though Java agents are platform–independent the agents require that the platforms have pre-installed compilers, and libraries to install the benchmarking software. Currently we are able to run the Surge tools only on Linux–based systems, because Surge–requiring thread libraries are unavailable on other systems.

In the workload dataset generation step, a monitoring agent executes a list of command–line programs with parameters specifying the number of files, maximum number of file reference, as required by SURGE, [1].

During the request generation and data analysis steps, a monitoring agent invokes Linux processes, which handle actual HTTP requests and data processing, and the agent waits until the processes finish. The monitoring agent checks the correct completion of the Linux processes by comparing the output strings with expected ones. We use Perl–written scripts to process the measurement data for efficiency and ease of use.

We now examine the `bondMultiAgentDeployStrategy`. Its function is to get from the model of the coordinator a vector containing the list of agents and then to go through this list and identify the host where the agent is expected to run, the path to the blueprint of the agents and Alias of the agent.

```
public class bondMultiAgentDeployStrategy extends
bondDefaultStrategy {
  boolean first = true;
  int index = 0;
  Vector agents;
  long interval;
public long action(bondModel m, bondAgenda ba) {
 if (first) {
      agents = (Vector)getFromModel("Agents");
      first = false; }
 if (index < agents.size()) {
      try {Thread.currentThread().sleep(interval);}
      catch(InterruptedException e) { }
      Object o = agents.elementAt(index++);
      if (o instanceof Vector) {
         Vector v = (Vector)o;
         putIntoModel("RemoteAddress", v.elementAt(0));
         putIntoModel("Blueprint",
                          "blueprint/"+v.elementAt(1));
         putIntoModel("Alias", "Agent"+index);
         transition("success"); return 0l;}
      else if (o instanceof Hashtable) {
         Hashtable h = (Hashtable)o;
         if (h.containsKey("RemoteAddress")) {
                   putIntoModel("RemoteAddress", h.get
                          ("RemoteAddress"), false);}
         else {transition("fail"); return 0l;}
         if (h.containsKey("Blueprint")) {
             putIntoModel("Blueprint",
                           h.get("Blueprint"), false);}
         else { transition("fail"); return 0l;}
         if (h.containsKey("Model")) {
          putIntoModel("Model", h.get("Model"), false);}
         if (h.containsKey("Alias")) {
           putIntoModel("Alias", h.get("Alias"), false);}
         transition("success"); return 0l;}
 }
 transition("next");return 0l;}
}

public class bondBarrierWaitStrategy extends
                 bondBarrierEnabledStrategy {
  private Object blocker = new Object();
  private bondBarrier barrier;
  final static long WAITTIME = 30000;
  public long action(bondModel m, bondAgenda ba) {
    // create barrier
    String bn = (String)getFromModel("BarrierName");
    String sa = (String)getFromModel("SpaceAddress");
    String sn = (String)getFromModel("SpaceName");
    int numToken = ((Integer)getFromModel
                        ("NumToken")).intValue();
    try {
      if (!createBarrier(bn, sa, sn, numToken, null,
        true)) { transition("fail");return 0l; } }
    catch (TupleSpaceException e) {
      transition("fail");
      return 0l; }
    // wait until wake-up from callback
    while (true) {
      try { synchronized (blocker) {
          blocker.wait(WAITTIME); } }
      catch (InterruptedException e) { }
      // compare number of token
      if (barrier != null) {
    if (barrier.goalReached()) {
      barrier = null;
      break;
    }
    else { barrier = null;}
      }
    }

    // make transition
    transition("success");
    return 0l;
```

}

At each step, failures are monitored by the distributed adaptive fault detection algorithm described in [33]; failures of an worker agents during benchmarking may lead to incomplete workload generation or loss of measurement data. The workers and the coordinator form a ring monitoring topology at each step. The monitoring topology is initialized at each step with a new set of worker agents. The coordinator agent is the initial contact point providing current list of fault–free agents and it responsible for fault recovery in the case of failure detection.

9.3.3 Agent-Based Workflow Management

Motivated by deficiencies of existing workflow management systems (WFMS) in the area of flexibility and adaptability to change we initiated work on building a workflow management framework on top of the Bond system [44]. Usually in WFMS implementations agents enhance the functionality of existing WFMS and act as personal assistants performing actions on behalf of the workflow participants and/or facilitating interaction with other participants or the workflow enactment engine. We propose an agent–based WFMS architecture in which software agents perform the core task of workflow enactment. In particular we concentrate on the use of agents as *case managers*: autonomous entities overlooking the processing of single units of work. Our assumption is that an agent–based implementation is more capable of dealing with dynamic workflows and with complex processing requirements where many parameters influence the routing decisions and require some inferential capabilities. We also believe that the software engineering of workflow management systems is critical and instead of creating monolithic systems we should assemble them out of components and attempt to reuse the components.

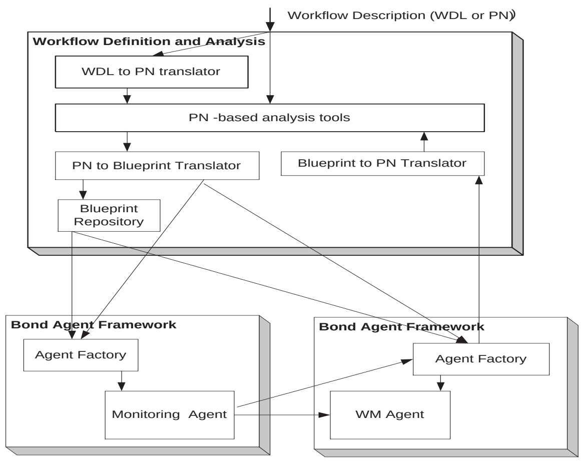


Fig. 9.31 Workflow management in Bond

Figure 9.31 illustrates the definition and execution of a workflow in Bond. The workflow management agent originally created from a static description can be modified based upon the information provided by the monitoring agent. Several workflows may be created as a result of mutations suffered by the original workflow. Once the new blueprint is created dynamically, it goes through the analysis procedure and only then it can be stored in the blueprint repository. The distinction between the monitoring agent and the workflow management agent is blurred, if necessary they can be merged together into a single agent.

We use Petri nets as an unambiguous language for specifying the workflow definition and provide a mechanism for enacting a large class of Petri net–based workflow definitions on the Bond finite state machine. For interoperatbility reasons we also supply a translator from the industry standard Workflow Process Definition Language [17] to our internal representation.

9.3.4 Other Applications

To test the limitations and the flexibility of our system, we developed several other applications of Bond agents ranging from a resource discovery agent to a network of PDE solver agents. We overview some of these applications.

9.3.4.1 Resource Discovery The Bond agents for resource discovery and monitoring have distinct advantages over statically configured monitors which have to be re-designed and programmed if they are deployed to other heterogeneous nodes. Moreover the local monitors should be pre-installed. The dynamic composability and surgery of the Bond agents makes it possible to deploy monitoring agents on the fly with strategies compatible with target nodes, and modify them on demand either to perform other tasks or to operate on other heterogeneous resources.

We developed an agent-based resource discovery and monitoring system shown in Figure 9.32. Agents running at individual nodes learn about the existence of other agents by using *distributed awareness*, a distributed mechanism by which each node maintains locations of other nodes it has communicated with over a period of time and exchanges periodically this information among themselves [31]. Whenever an agent, a *beneficiary agent*, needs detailed information about individual components of other nodes, it uses the distributed awareness information to identify a target node, then creates a blueprint of a monitoring agent capable of probing and reporting the required information on the target node, and sends the blueprint to an agent factory of it. The agent factory assembles the monitoring agent and launches it to work. A blueprint repository, which is either local or remote, stores a set of strategies. By sending a surgery script, the beneficiary agent can modify the agents as desired.

This solution is scalable and suitable for heterogeneous environments where the architecture and the hardware resources of individual nodes differ, the services provided by the system are diverse, the bandwidth and the latency of the communication links cover a broad range. On the other hand, the amount of resources used by agents might be larger than those required by other monitoring systems.

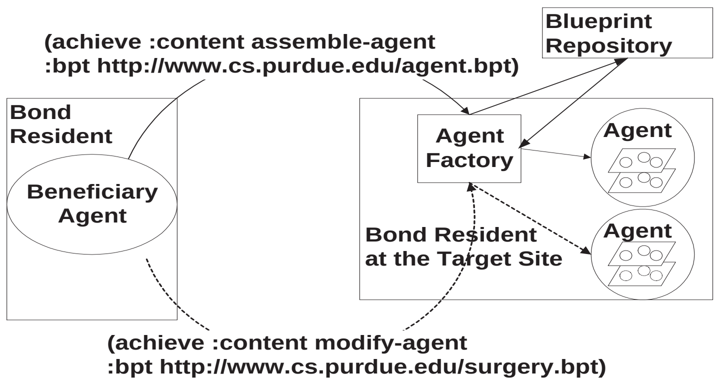

(achieve :content assemble-agent
:bpt http://www.cs.purdue.edu/agent.bpt)
Blueprint Repository
Bond Resident
Beneficiary Agent
Agent Factory
Agent
Agent
Bond Resident at the Target Site
(achieve :content modify-agent
:bpt http://www.cs.purdue.edu/surgery.bpt)

Fig. 9.32 The dynamic deployment and modification of monitoring agents. The Beneficiary agent sends either a blueprint (solid line) or a surgery script (dotted line) to an agent factory to deploy a monitoring agent or to modify an existing one. The agent factory assembles it with local strategies or ones from a remote blueprint repository

9.3.4.2 A Network of PDE Solver Agents. Data parallelism is a common approach to reduce the computing time and to improve the quality of the solution for data-intensive applications. Often the algorithm for processing each data segment is rather complex and the effort to partition the data, to determine the optimal number of data segments, to combine the partial results, to adapt to a specific computing environment and to user requirements must be delegated to another program. Mixing control and management functions with the computational algorithm leads in such cases to brittle and complex software. We developed a network of PDE solver agents and discussed its application for modeling propagation and scattering of acoustic waves in the ocean [51].

Agents with inference abilities coordinate the execution and mediate the conflicts while solving PDEs. Three types of agents are involved: one `PDECoordinator` agent, several `PDESolver` and `PDEMediator` agents. The `PDECoordinator` is responsible with the control of the entire application, a `PDEMediator` arbitrates between the two solvers sharing a boundary between two domains, and a `PDESolver` is a wrapper for the legacy application. Thus we were able to identify with relative ease the functions expected from each agent and write new strategies in Java. The actual design and implementation of the network of PDE solving agents took less than one month. Thus, the main advantage of the solution we propose is a drastic reduction of the development time from several months to a few weeks.

9.4 FURTHER READING

The Ph.D. dissertations of Ladislau Bölöni [2] and Kyungkoo Jun [30] describe in detail various components of the system. The first, covers the distributed object system and the agent framework, the second the extensions to the system and applications.

An overview of the system is presented in [5] and more details can be found in [9]. The subprotocols are discussed in [4], security aspects are presented in [24] and [23]. The agent model is presented in [6] and [7] and the surgery in [8].

Applications of the system are presented as follows: multimedia applications in [32] and [56], resource discovery in [31], the workflow management system in [44], the network of PDE solvers in [10] and [51], applications to problem solving environments in [36], monitoring of web servers in [34]. An algorithm for fault detection and fault information dissemination can be found in [33].

Tuplespaces are presented in several papers related to Linda [12], [13], Javaspaces [54], Pagespace [15], Jada, [16], T Spaces, [28], [49].

A number of Java-based agent or distributed object systems are presented in [21], [53]. The Java expert system shell, Jess is discussed in [20]. Java security is surveyed in [22]. Excellent references for mixins and design patterns are [11] and [18].

There is a vast literature on Web monitoring, [1], [26], [38], [40], [41], [46], [50].

A discussion of biological metaphors applied to the design on complex systems can be found in [3] and [37]. Reference [35] discusses applications of mobile agents for process coordination on information grids.

9.5 ACKNOWLEDGMENTS

The system described in this chapter was developed in the Bond laboratory of the Computer Sciences Department at Purdue University in the 1998-2001 time frame. Several postdoctoral fellows and former graduate students have contributed to the system, they are: Dr. Ladislau Bölöni, now with Cplane, Dr. Kyungkoo Jun, now with Intel Corporation, and Dr. Ruibing Hao, now with Bell Labs. Kryzsztof Palacz and Radu Sion have also been associated with the project.

REFERENCES

1. P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proc. SIGMETRICS 98*, June 1998.

2. L. Bölöni. Contributions to Distributed Objects and Network Agents, April 2000. Ph.D. Dissertation, Department of Computer Sciences, Purdue University.

3. L. Bölöni, R. Hao, K. Jun, and D. C. Marinescu. Structural Biology Metaphors Applied to the Design of a Distributed Object System. In *Proc. Workshop on*

Biologically Inspired Solutions to Parallel Processing Problems, LNCS, pages 275–283. Springer Verlag, 1999.

4. L. Bölöni, R. Hao, K. K. Jun, and D. C. Marinescu. An Object-Oriented Approach for Semantic Understanding of Messages in a Distributed Object System. In *Proc. Int. Conf. on Software Engineering Applied to Networking and Parallel/ Distributed Computing, Rheims, France*, May 2000.

5. L. Bölöni, K. Jun, K. Palacz, R. Sion, and D. C. Marinescu. The Bond Agent System and Applications. In *Proc. 2nd Int. Symp. on Agent Systems and Applications and 4th Int. Symp. on Mobile Agents (ASA/MA 2000)*, volume 1882 of *LNCS*, pages 99–112. Springer Verlag, 2000.

6. L. Bölöni and D. C. Marinescu. A Component Agent Model - from Theory to Implementation. In *Proc. Second Intl. Symp. From Agent Theory to Agent Implementation*, pages 633–639. Austrian Society of Cybernetic Studies, 2000.

7. L. Bölöni and D. C. Marinescu. A Multi-Plane Agent Model. In *Autonomous Agents, Agents 2000*, pages 80–81. ACM Press, 2000.

8. L. Bölöni and D. C. Marinescu. Agent Surgery: The Case for Mutable Agents. In *Proc. Workshop Biologically Inspired Solutions to Parallel Processing Problems*, volume 1800 of *LNCS*, pages 578–585. Springer Verlag, 2000.

9. L. Bölöni and D. C. Marinescu. An Object-Oriented Framework for Building Collaborative Network Agents. In H.N. Teodorescu, D. Mlynek, A. Kandel, and H.-J. Zimmerman, editors, *Intelligent Systems and Interfaces*, Int. Series in Intelligent Technologies, chapter 3, pages 31–64. Kluwer Publising House, 2000.

10. L. Bölöni, D. C. Marinescu, P. Tsompanopoulou J.R. Rice, and E.A. Vavalis. Agent-Based Networks for Scientific Simulation and Modeling. *Concurrency Practice and Experience*, pages 845–861, 2000.

11. G. Bracha and W. Cook. Mixin-Based Inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, October 1990. ACM Press.

12. N. Carriero and D. Gelernter. Linda in Context. *Comm. of the ACM*, 32(4):444–458, April 1989.

13. N. Carriero, D. Gelernter, and J. Leichter. Distributed Data Structures in Linda. *ACM Trans. on Programming Languages and Systems*, 8(1), Jan 1986.

14. K. M. Chandy, J. Kiniry, A. Rifkin, and D. Zimmerman. Infosphere Infrastructure User's Guide. URL `http://www.infospheres. caltech. edu`, January 1998.

15. P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. *Computer Networks and ISDN Systems*, 28(7-11):941–952, 1996.

16. P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag: Heidelberg, Germany, April 1997.

17. Workflow Management Coalition. Interface 1: Process Definition Interchange Process Model, 11 1998. WfMC TC-1016-P v7.04.

18. E.Grama, R. Helm, R. Johnson, and J.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Inc, 1995.

19. T. Finin, R. Fritzon, D. McKay, and R. McEntire. KQML - A Language and Protocol for Knowledge and Information Exchange. In *Proc. 13th Int. Workshop on Distributed Artificial Intelligence*, pages 126–136, Seatle, WA, Jul 1994.

20. E. Friedman-Hill. Jess, the Java Expert System Shell. Technical Report SAND98-8206, Sandia National Laboratories, 1999.

21. G. Glass. ObjectSpace Voyager — The Agent ORB for Java. *Lecture Notes in Computer Science*, 1368:38–47, 1998.

22. L. Gong. Java Security Architecture (JDK 1.2). Technical report, JavaSoft, July 1997.

23. R. Hao, L. Bölöni, K. Jun, and D. C. Marinescu. An Aspect-Oriented Approach to Distributed Object Security. In *Proc. Fourth IEEE Symp. Computers and Communication, ISCC99*, pages 23–31. IEEE Press, 1999.

24. R. Hao, K. Jun, and D. C. Marinescu. Bond System Security and Access Control Models. In *Proc. IASTED Conference on Parallel and Distributed Computing*, pages 520–524. Acta Press, 1998.

25. D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of State Charts. In *Proc. 2nd Symp. on Logic in Computer Science (LICS 87)*, pages 54–64. IEEE Computer Society Press, 1987.

26. Holistix. Holistix. URL `http://www.holistix.net`.

27. J Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th International Python Conference*, San Jose, California, October 1997.

28. Ibm. TSpaces: Intelligent Connectionware. `www.almaden.ibm.com`.

29. Ibus. URL `http://www.softwired-inc.ch` .

30. K Jun. Monitoring and Control of Networked Systems with Mobile Agents: Algorithms and Applications, April 2001. Ph.D. Dissertation, Department of Computer Sciences, Purdue University.

31. K. Jun, L. Bölöni, K. Palacz, and D. C. Marinescu. Agent–Based Resource Discovery. In *Proc. Heterogeneous Computing Workshop 2000*, pages 43–52. IEEE Press, 2000.

32. K. Jun, L. Bölöni, D. Yau, and D. C. Marinescu. Intelligent QoS Support for an Adaptive Video Service. In *Proc. IRMA 2000 - Challenges of Information Technology Management in the 21st Century*, pages 1096–1098. Idea Group Publishers, 2000.

33. K. Jun and D. C. Marinescu. An Algorithm for Fault Detection and Fault Information Dissemination for Federations of Mobile Agents. Submitted.

34. K. Jun and D. C. Marinescu. Monitoring and Adaptive Control of Web Servers with Mobile Agents. Submitted.

35. D. C. Marinescu. Reflections on Qualitative Attributes of Mobile Agents for Computational, Data, and Service Grids. In *Workshop on Agent-Based Cluster and Grid Computing 2001*. IEEE Press, 2001.

36. D. C. Marinescu and L. Bölöni. A Component-Based Architecture for Problem Solving Environments. *Mathematics and Computers in Simulation*, pages 279–293, 2001.

37. D. C. Marinescu and L. Bölöni. Biological Metaphors in the Design of Complex Software Systems, journal = Journal of Future Computer Systems. 17:345–360, 2001.

38. Service Metrics. Service Metrics. URL `http://www.servicemetrics.com`.

39. Sun Microsystems. Java Developer Connection. `http://java.sun.com`.

40. Mindcraft. WebStone 2.5. URL `http://www.mindcraft.com/webstone/`.

41. D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. In *Proceedings of Internet Server Performance Workshop*, pages 59–67, June 1998.

42. OMG. *The Common Object Request Broker : Architecture and Specification. Revision 2.3.* TC Document 99-10-07, October 1999.

43. Orbix. URL `http://www.iona.com/`.

44. K. Palacz and D. C. Marinescu. An Agent-Based Workflow Management System. In *Proc. AAAI Spring Symp. Workshop "Bringing Knowledge to Business Processes"*, pages 119–127. AAAI Press, 2000.

45. R. Sethi and J. D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM*, 4(17):715–728, October 1970.

46. SPECweb99. The Standard Performance Evaluation Corporation. SPECweb99. URL `http://www.specbench.org/osg/web99/`.

47. Simon St. Laurent. *XML: a primer*. IDG Books, San Mateo, CA, USA, second edition, 1999.

48. Sun Microsystems. Java RMI.

49. L. Tobin, M. Steve, and W. Peter. T Spaces: The Next Wave. *IBM System Journal*, 37(3):454–474, 1998.

50. tpc. TPC Benchmark W (TPC-W). URL `http://www.tpc.org/wspec.html`.

51. P. Tsompanopoulou, L. Bölöni, D. C. Marinescu, and J. R. Rice. The Design of Software Agents for a Network of PDE Solvers. In *Workshop on Agent Technologies for High Performance Computing, Agents 99*, pages 57–68. IEEE Press, 1999.

52. J. Viega, P. Reynolds, W. Tutt, and R. Behrends. Multiple Inheritance in Class Based Languages. Technical Report 03, University of Virginia, 1998.

53. Visibroker. URL `http://www.borland.com/visibroker/` .

54. J. Waldo. JavaSpace Specification - 1.0. Technical report, Sun Microsystems, March 1998.

55. WebBench. URL `http://www.zdnet.com`.

56. D. Yau, K. Jun, and D. C. Marinescu. Middleware QoS Agents and Native Kernel Schedulers for Adaptive Multimedia Services and Cluster Servers. In *Proc. Real-Time System Symp. 99*. IEEE Press, 1999.