A Gentle Introduction to Bond

November 2, 1998

Contents

1 Overview **1**

2 Terms and concepts **3**

3 Architecture **5**
- 3.1 Objects and containers 5
- 3.2 Communication fabric 5
- 3.3 Probes, an aspect-oriented approach to complex object design 7

4 Agent Framework **11**
- 4.1 Monitoring Framework 12

5 Security Framework **14**

6 Bond Programing **16**
- 6.1 Property access, *set* and *get* 17
- 6.2 Messages, `say` 18
- 6.3 Locating Bond objects 19
- 6.4 Pairing of messages, `needReply()`, `onReply()`, and `replyWith()` 19
- 6.5 Asynchronous communication in Bond: `waitReply()` and `ask()` 20

7 Demos **21**
- 7.1 StockWatcher/Client: Data Source Monitoring using the Bond Monitoring Framework 21

1 Overview

This document introduces Bond, a Java-based, object-oriented middleware for network computing. Network computing is a paradigm that emphasises the use of network resources, computing resources distributed across the network, over local resources. Network resources are hosts, programs and data. Middleware is a software layer that allows developers to mold systems tailored to specific needs from components and develop new components based upon existing ones. The goals of the Bond system are to:

- facilitate access to network resources,
- support collaborative activities, and
- accomodate software diversity and hardware heterogeneity.

The basic components of Bond are meta-objects and agents. Meta-objects provide synthetic information about network resources and agents use this information to access and manipulate network objects on behalf of users. The basic philosophy of the Bond middleware is to allow developers and users to define meta-objects and agents using the *frameworks* provided by the system.

Bond can be used in stand-alone mode or to connect to a Bond domain. A Bond domian consists of a society of users with common interests and a number of services. In stand-alone mode a user may create meta-objects and agents and execute locally several demos. To create a Bond domain one needs to activate the servers provided with the package.

An application of the Bond middleware is a Virtual Laboratory, an environment for controlling experiments and combining experimental data with computer simulations. Some of the functions desirable in such an environment are:

- System-level resource management,
- User-level resource management,
- Annotation of network resources,
- Management of communication channels,
- Support for collaborative work, ability to share resources,
- Access to computational servers,
- Remote start-up and execution of single programs,
- Execution of workflows,
- Persistent storage,
- Data migration,
- Scheduling,
- Monitoring,
- Access control and security related functions,
- Accounting.

Bond is a message-oriented system, it uses KQML [6], as a meta-language for inter-object communication. KQML offers a variety of message types (performatives) that express an attitude regarding the content of the exchange. Performatives can also assist agents in finding other agents that can process their requests. A performative is expressed as an ASCII string, using a Common Lisp Polish-prefix notation. The first word in the string is the name of the performative, followed by parameters. Parameters in performatives are indexed by keywords and therefore order independent.

There are a number of efforts to develop middleware for distributed object computing. The Common Object Request Broker Architecture (CORBA) [3] is the most influential. Microsoft, has its own competing architecture Distributed Component Object Model (DCOM).

2 Terms and concepts

- A *Bond Object* is collection of data fiels and methods. The data component of an object consists of several *fields* defined at compile-time and a vector of *dynamic properties* defined at run-time. Each object has a *unique bondId.*
- A *container* is a Bond object that encapsulates a collection of objects. There are two types of containers, active and passive. Residents and workspaces are examples of containers.
- A *component* is a persistent object with a visual interface that can be manipulated with graphical toolkits and can communicate with other components, either locally or remotely, via one or more mechanisms including message passing, procedure calls, etc.
- The term *network resources* refers to hardware, programs and data sets.
- A *meta-object* is a Bond object providing information about one or more network resources. A meta-object contains links to the network object(s), either the URL of a program or data file or the netId of a host.
- A *core object* is one implementing the basic functionality of the Bond system.
- A *resident* is an active container consisting of several threads of control, a local directory and a set of objects.
- An *workspace* is a passive container consisting of several, possibly many objects belonging to one user.
- *Message Oriented Middleware, MOM.* System whose components communicate only by means of messages, they do not relay on remote method invokation.
- A *message* is a Bond object. Messages are used to implement remote data access and remote procedure calls. Bond objects communicate using string messages. Every message is a sentence in the KQML [6] agent communication language.
- An *active message* is one delivered immediately upon receipt to its destination. A passive message is queued in a buffer for later delivery.
- A *shadow object* is a proxy for a remote object. A shadow is an abstraction allowing distant objects to communicate with one another as if they were co-located.
- A *virtual network* is a local collection of shadow objects.
- A *waiting slot* is an abstraction to pair a request with a replay (see the `onReply` method that creates a waiting slot.
- A *subprotocol* is a closed subset of KQML messages, a specialized language needed to perform a specific task. Each Bond message is stamped with the subprotocol it belongs to.
- *Subprotocol inheritance.* An object can only understand subprotocols implemented by its own class or inherited from its ancestors.
- A *probe* is an object that understands one or more specialized subprotocols and can be attached to an existing object. The set of the dynamic properties of an object is interogated only if neither the object nor its ancestors are capable to understand the subprotocol.

- A *preemptive probe* is a probe that has precedence in handling a message. A preemptive probe is invoked before checking the subprotocol inheritance while a normal probe is activated after checking the subprotocol inheritance.
- A *Bond domain* is a community of users collaborating towards solving a problem and a set of shared resources.
- A *server* is a complex object performing a well defined function in a Bond domain.
- A *Persistent Storage Server* is a shared system resource capable to store and retrive workspaces and individual objects belonging to the users in a Bond domain. Note that an workspace can be saved locally as well.
- The *System Monitor* is responsible for starting up all servers in a Bond domain and for replicating them when necessary.
- A *Directory Server* is a shared resource in a domain capable to provide information about remote objects. When starting up a resident should register itself with a Directory Server in that domain.
- *KQML* The Knowledge Query and Manipulation Language supports communication between software agents.
- An *agent* is a program that pursue its agenda autonomously and is able to cooperate with other agents towards reaching its goal.
- The *state machine* of an agent defines the states and the transitions among the states implemented by the agent.
- The *model* is a view of the world as seen by an agent. The model consists of a set of variables describing the state of the outside world.
- The *strategy* associated with a state of an agent is a program executed by the agent when it enters that state.
- The *access control* mechanism defines the set of messages an object is willing to receive from other objects. Since Bond is a pure message-oriented middleware, actions on remote objects occur only as side-effects of messages.
- *Authentication* is the process of veryfing the identity of a remote object.
- *Delegation* is a security paradigm confering an object the same security and access control priviledges the creator of the object enjoys.
- The *security context* is a set of objects necessary to implement a security probe.
- A *secure object* is one with a security probe attached to it.
- A *security ticket* is an object that contains the proof of access rights for a resource.
- A *security certificate* is a proof of the binding between the userId and the public key of the user.
- A *monitor* is an object that collects information about the properties of other objects, analyze the information gathered through monitoring, and takes actions based on the result of this analysis.

- A *subject* is an object whose properties are of interest to other objects and it is willing to provide them upon request.
- *Notification model* defines the notification message pattern. Possible models are *time-triggered*, *change-triggered*, and *hybrid*.
- *Subscription-based monitoring* is a monitoring model in which a monitor subscribes to a list of properties of a subject, which in turn notifies the monitor of the property changes according to one of notification models.
- A *monitoring probe* is a dynamic object implementing the monitoring sub-protocol. It is attached to other objects to add monitoring functionality.
- *Monitoring interfaces* are a list of mandatory methods that the object integrating the monitoring probe should implement. The methods are invoked by the monitoring probe.

3 Architecture

3.1 Objects and containers

A *Bond Object* is collection of data fiels and methods, [1]. The data component of an object consists of several *fields* and a vector of *dynamic properties*. Each object has a *unique bondId*. The granularity of Bond objects ranges from simple objects, e.g. a message to complex objects like a directory server. Some objects are *passive*, e.g. a message or a meta-object associated with a data file, others are *active* they have one or more threads of control. Active as well as passive objects can communicate using the `say()` method. All objects are persistent. A segment of the Bond object hierarchy is presented in Figure 1. The full Bond object hierarchy is presented elsewhere.

A *container* is a Bond object that encapsulates a collection of objects. A message for a remote object is always delivered to the container where the object is located and the messaging thread together with the local directory cause the delivery of the message to the local object. There are two types of containers, *active and passive*. Residents and workspaces are examples of containers. A *resident* is an active container consisting of several threads of control, a local directory and a set of objects. Bond objects with the exception of *light-weight objects* like meesages and shadows always register with the local directory upon creation.

Figure 2 shows the structure of a resident. The messaging thread of a resident delivers messages from remote objects to local ones and sends out messages originating from local objects. Several other threads may be running as a side-effect of messages in sub-protocols understood by the resident. An *workspace* is a passive container consisting of several, possibly many objects belonging to one user. An workspace can be saved to and retrived from persistent storage (or stored on the local host and retrived by the resident).

3.2 Communication fabric

The system provides a communication fabric allowing objects to interract with one another. The communication fabric is based upon *active messages*. An *active message* is one delivered immediately upon receipt to its destination. A passive message is queued in a buffer for later delivery. Bond favors non-blocking communication, typically the sender continues execution execution after sending a request and it is interrupted upon the arrival of the reply. There are means to block, waiting for a message(see the `waitReply()` function).

A *shadow object* is a proxy for a remote object. A shadow is an abstraction designed to allow distant objects to communicate with one another as if they were co-located. A message

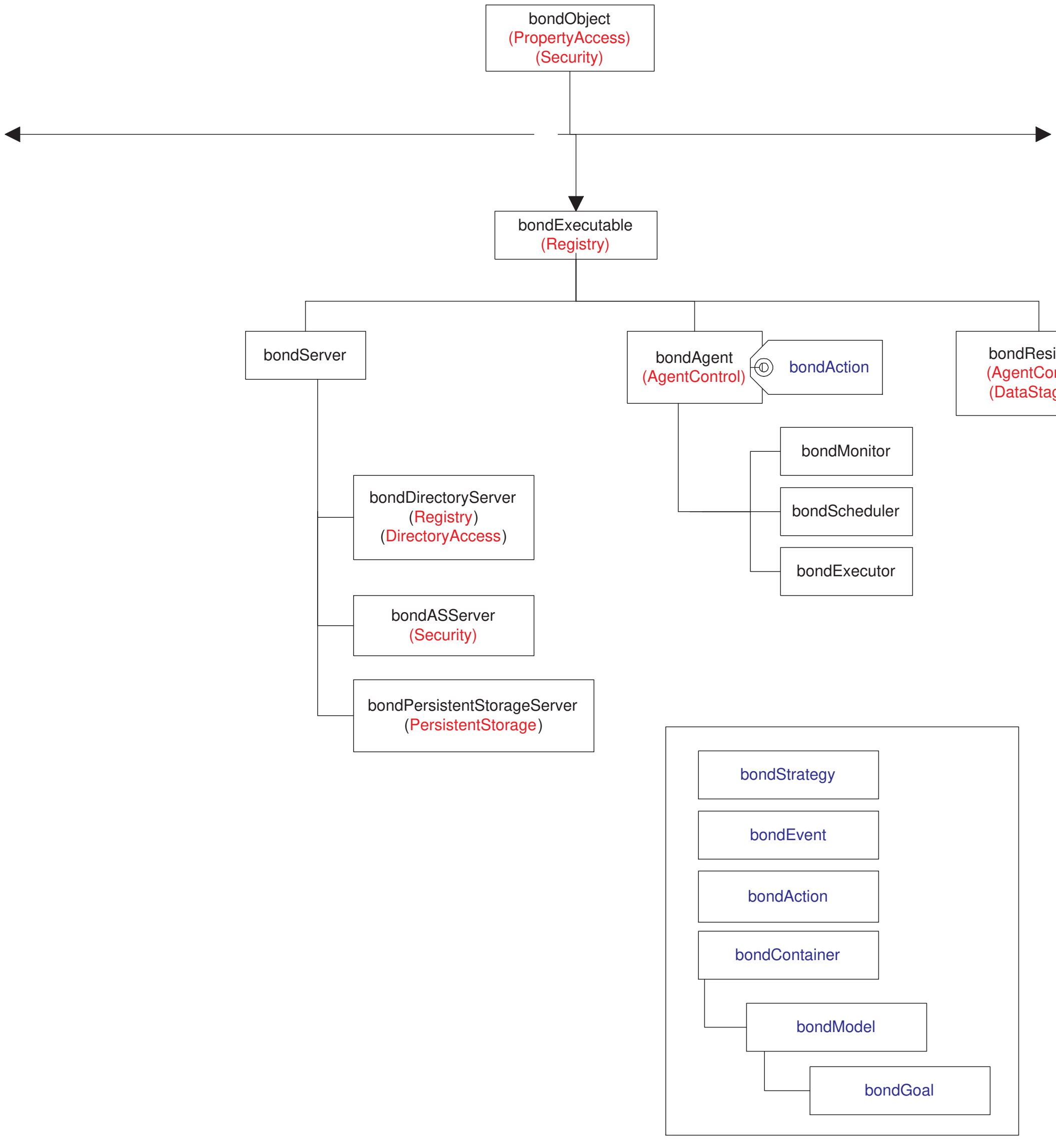

Figure 1: A segment of the Bond object hierarchy and the subprotocols understood by various classes.

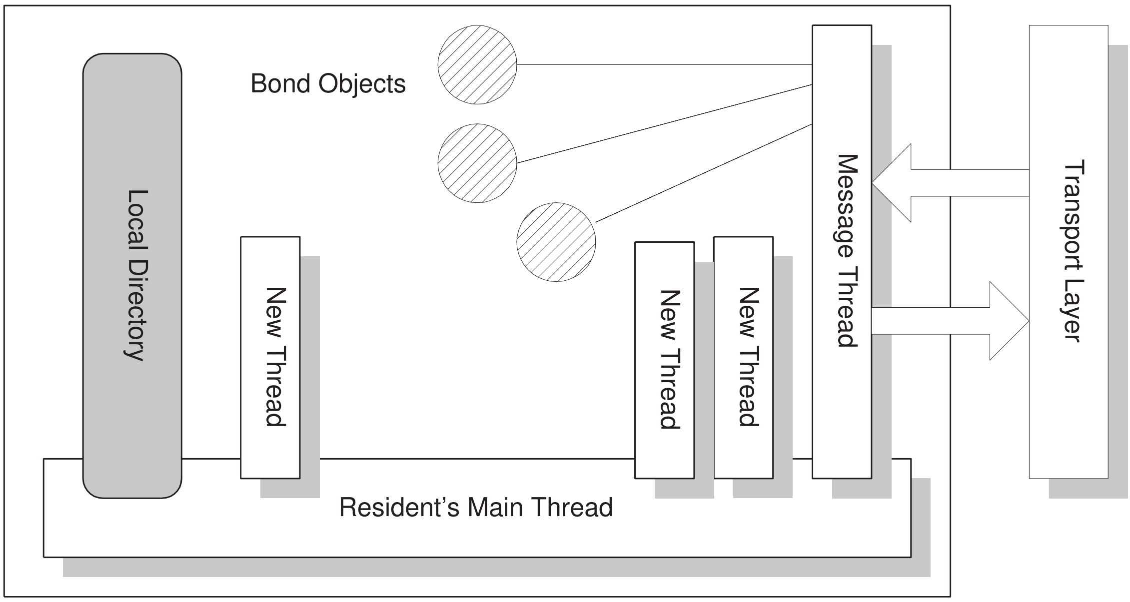

Figure 2: A Bond resident. All objects in the container, excluding shadows are registered with the local directory. The messaging thread of the resident delivers messages to local objects identified by their bondId. Messages for remote objects are sent out by the messaging thread using the address of the remote container provided by the local shadow. The main thread of a resident may start up new threads as requested by various sub-protocols. A resident could be configured as a server or have one or more agents.

delivered to the local shadow is guaranteed to reach the remote object. Shadow objects are the conceptual equivalents of the stubs in Corba and RMI and can be implemented using stubs. Shadows are the only component of the Bond system that depends upon the underlying transport mechanism. Porting a Bond networking solution to a new transport middleware requires only to re-implement the `bondShadow` object, while all the other components can be reused even without being recompiled.

A *virtual network* is a local collection of shadow objects. This abstraction is necessary to create dynamic sets of semantically related objects. For example, a directory maintains a virtual network of local directories, a scheduler agent maintains a virtual network of execution agents.

A *subprotocol* is a closed subset of KQML messages, a specialized language needed to perform a specific task. The subset is closed becuase a subprotocol does not reference outside messages, the reply is a member of the same subprotocol with the question. Examples of generic Bond subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol, see 3.2. This abstraction is necessary to introduce a *structure in the semantic space of the messages.* An object can only understand sub-protocols implemented by its own class or inherited from its ancestors.

3.3 Probes, an aspect-oriented approach to complex object design

Bond relies on *probes* to separate various aspects of complex object design. A *probe* is an object that understands one or more specialized subprotocols and can be attach to an existing object. Security and monitoring probes can be attached to an existing object. Security probes implement several security and access control models. Monitoring probes implement the Bond

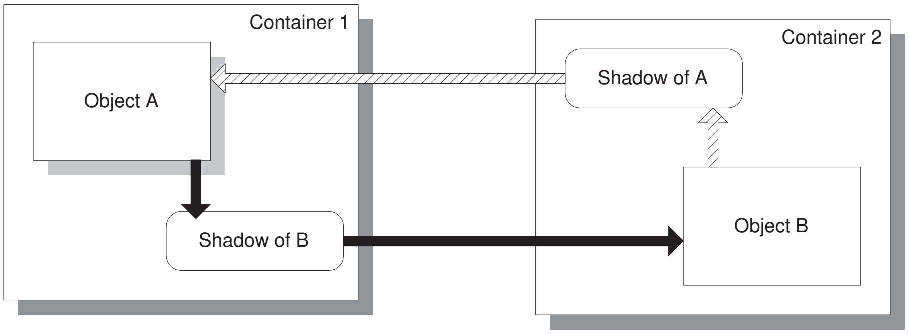

Figure 3: Communication between remote objects using their shadows. A shadow is a proxy for a remote object. To communicate with object B in container 2, object A in container 1 creates a shadow of B using the `find` function. Once the shadow of B is created, A applies the local `say()` method to send messages to it; then the shadow delivers the message to B. To send a message to A, B follows the same same procedure, creates a shadow of A then sends messages for A to its shadow.

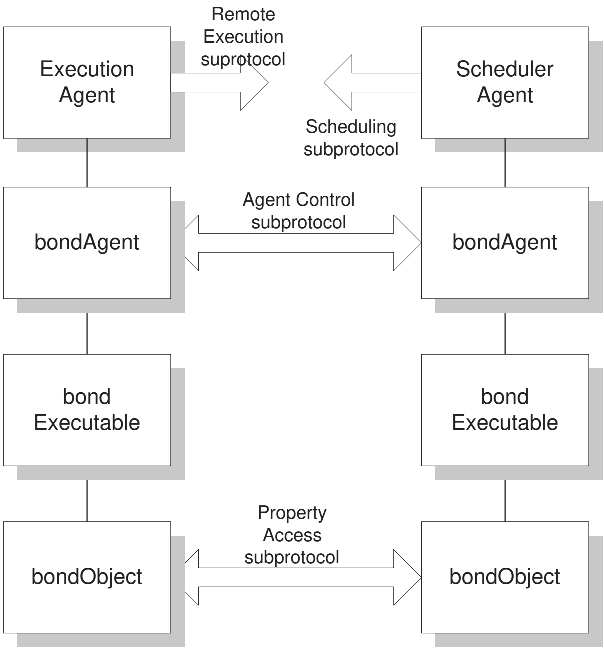

Figure 4: Subprotocol inheritance. A scheduling agent and execution agent understand the property access and the agent control subprotocols. In addition each one understands a specialized subprotocol related to its own functionality.

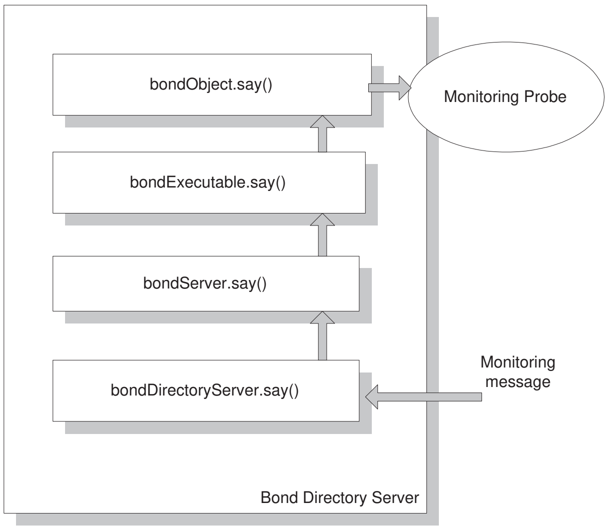

Figure 5: Probes. A `bondDirectoryServer` object does not understand the monitoring subprotocol. A monitoring message received by the `bondDirectoryServer` object message is passed to its ancestors, `bondServer, bondExecutable, bondObject`, but none of them understands the subprotocol. Then the dynamic properties of the object reveal that a monitoring probe is attached and the message is delivered to it. The monitoring probe understands the monitoring subprotocol and it is able to monitor other objects in behalf of the `bondDirectoryServer` or monitor the `bondDirectoryServer` in behalf of other objects.

monitoring model. They alow objects to continously monitor remote objects and be informed when changes of the properties of remote objects occur see Figure 3.2.

The system includes agent and meta-object factories and frameworks for monitoring and object security. The agent factory is capable to generate agents based upon blueprints describing the state-machine implemented by the agent,the strategies associated with every state, the agenda and the model of the world used by the agent.

Bond servers support the basic functionality necessary for collaborative activities. A Bond domain is a society of users with common interests. A system monitor starts up the servers and ensures fault-tolerance. Directory Servers allow objects to locate remote objects. The Persistent Storage Server is a repository for *user workspaces.*

Figure 6 shows the architecture of the system and its major components:

- The transport layer - ensures message delivery to remote objects. At this time Bond uses the Infospheres [4], to transport messages between Bond sites.
- Core elements - provide the communication fabric, containers and core objects.
- Frameworks - provide the infrastructure to build agents and meta-objects and augment them with monitoring, and security features.
- Extensions - provide blueprints for agents, metaobjects, probes and demos,
- Servers - directory, persistent storage, the system monitor, the QoS monitor, authentication, versioning, are some of the servers that can be activated in a Bond domain.
- Interfaces - GUIs and APIs provide user access to the system.

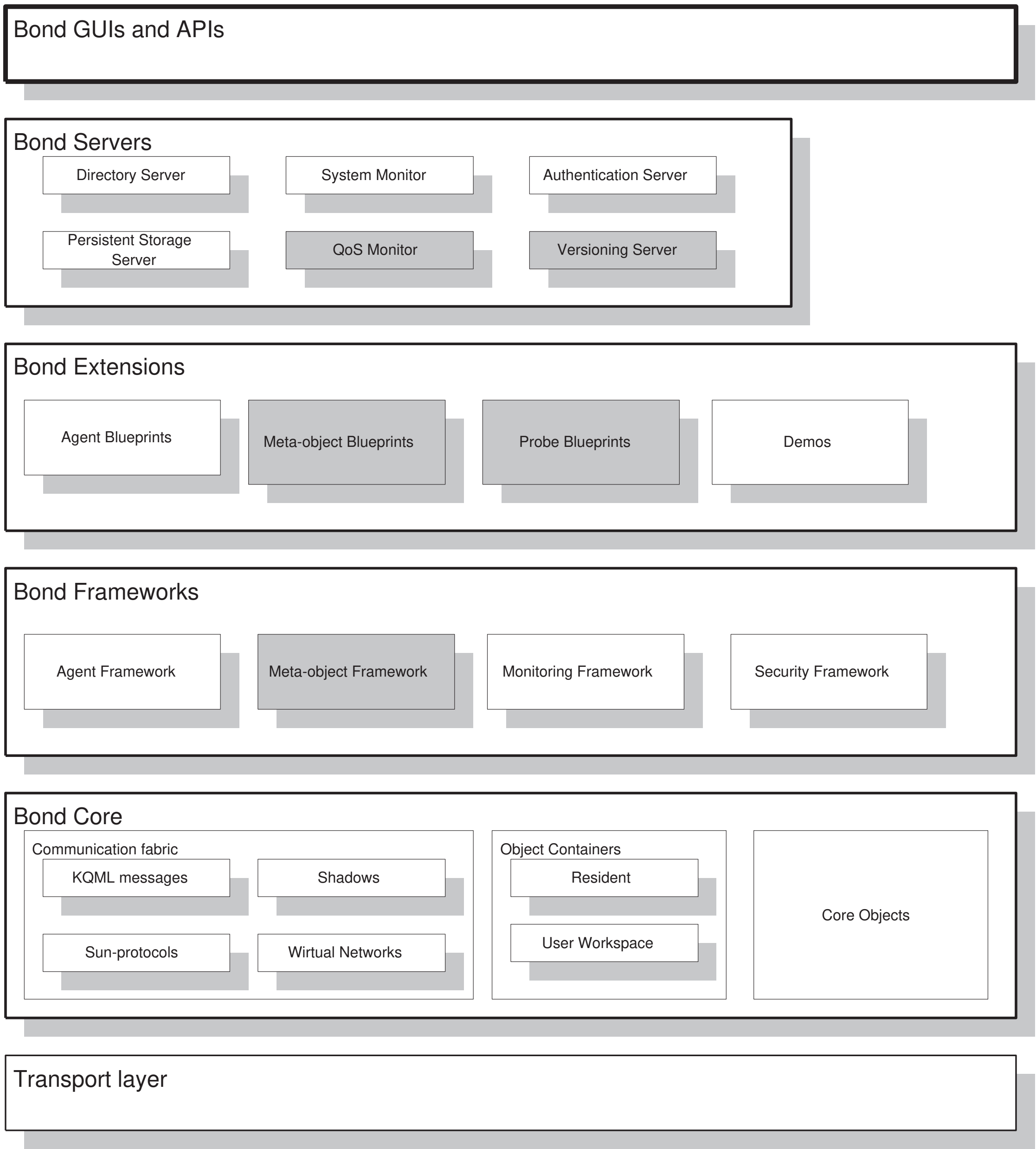

Figure 6: The architecture of the Bond middleware. The transport layer is provided by the info.net class of the Infospheres. Solid boxes are componentens not yet implemented. Bond servers are necessary to guarantee services in a Bond domain. Bond objects may communicate with each other even without the assistance of servers.

4 Agent Framework

Agents are programs that autonomously pursue their own agenda and are able to cooperate with other agents to accomplish their tasks.

The agent framework in Bond allows a programmer to simplify the programming of an agent because most of the functionality of the agents are already implemented. By default, Bond agents have the possibility to be controlled remotely and to cooperate with each other.

The task of an application programmer is limited to specify the agenda, the finite state machine of the agent, and the strategies associated with each state. Bond also provides a large database of ready-made strategies, so in typical cases a Bond agent can be assembled without programming.

The segment of the Bond object hierarchy related to the implementation of the agent framework is shown in Figure 7.

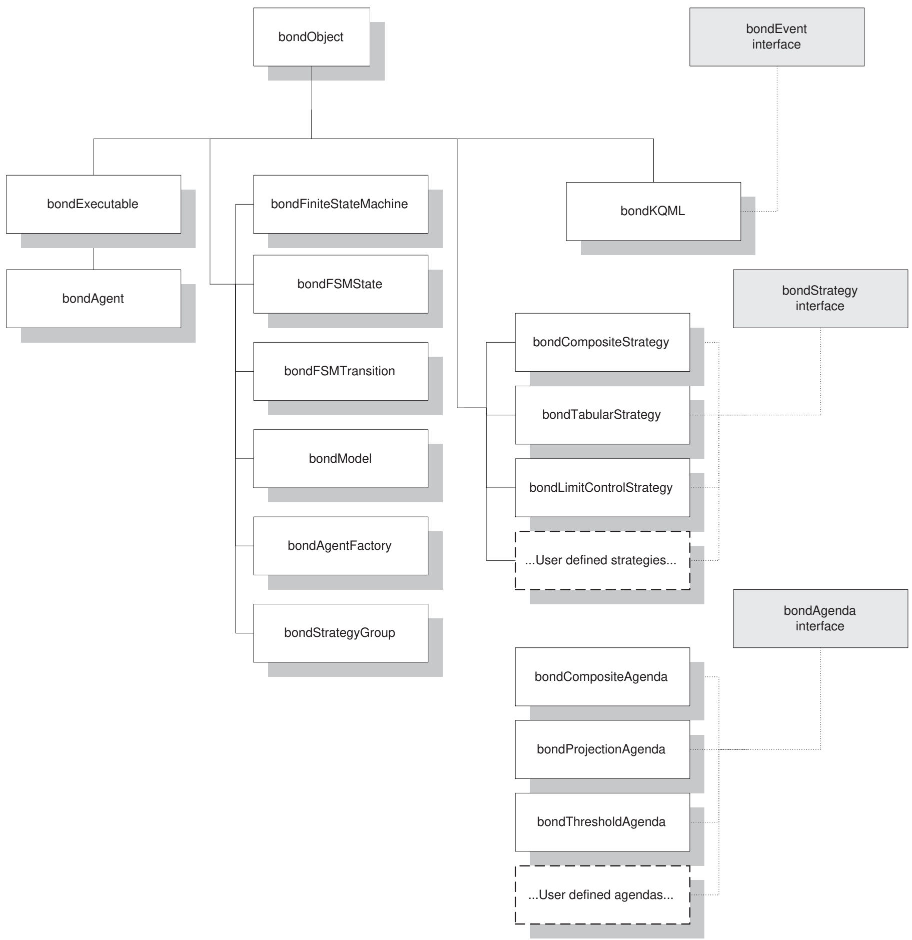

Figure 7: The segment of the Bond object hierarchy directly involved in creating agents

Events are object that implement the `bondEvent` interface, however all the events currently used in Bond agents are KQML messages, represented by the `bondKQML` object. The programmer may define new types of events.

The finite state machine is implemented by the `bondFiniteStateMachine`, `bondFSMState` and `bondFSMTransition` classes. These classes are general enough that they don't have to be overwritten for any agent covered by this formalism. A `bondFiniteStateMachine` can be constructed dynamically and saved or restored from the persistent storage.

Strategies are defined by any object which implement the `bondStrategy` interface. The model of the world is represented by the `bondModel` object. The information is represented in form of objects attached as dynamic properties of the model object. This approach allows the programmer complete freedom to use any knowledge representation approach he or she wants. We found it usefull to impose certain *conventions* on the representation which allows various strategies to work together. A programmer may or may not follow these conventions.

The agenda is an object implementing a `bondAgenda` interface. The `float distance(bondModel)` function should return the distance from the desired state of the world, while the `satisfiedBy()` function returns a boolean value showing if the agenda is satisfied or not.

One particular way of implementing the agenda is when the desired region is a projection of the Model, i.e. some values of the model should match specific values, while others can have arbitrary values. This is implemented in the `bondProjectionAgenda` object. Another possible agenda is when the desired region is an n-dimensional rectangle: we provide specific limits for the values of certain properties. This agenda is implemented by the `bondRectangleAgenda` object.

The `bondCompositeAgenda` object implements an *agenda calculus* supporting boolean expressions on ready made agendas. The ready-made agendas cover most of the usefull agendas one would think of using within the agent framework. These agendas can be created dynamically, without the need to write Java code.

Actions in the Bond system are object that implements the `bondAction` interface. Examples of defined actions in the object hierarchy are the `bondMethodCallAction` that allows a particular method of an object to be designated as an action and `bondAgent` which specifies that starting a new agent can be an action. The user can also define custom actions.

Strategies are object implementing the `bondStrategy` interface, which defines the `nextAction()` function. Strategies can be created using their constructors, created as part of strategy groups, or saved and loaded from persistent storage.

Figure 8 illustrates the states and the transitions of a data staging and remote execution agent. The strategy associated with every state is a sequence of actions, often the result of the execution of a program, or a script. For example the strategy for the "Data staging state" may consist of a script invoking the `ftp` file transfer protocol, the strategy for "Starting legacy program" state invokes a system call to start up a process running the program. The model of this agent consists of variables to store information needed by various strategies, e.g. the path to the executable for the legacy application, the completion code of the strategy for each state, and so on.

4.1 Monitoring Framework

Objects in Bond are cooperative. They provide and request services one another. They are inherently dynamic. Agent objects appear on demand and disappear. Permanent servers change their state. It is important for objects to know the state of other objects and their changes.

We design a *subscription-based monitoring model* [5]. The following terms are used to describe the model:

- *Property:* an entity consisting of name and its value.
- *Event:* the changes of properties.
- *Subject:* an object whose properties are of interest to other objects and it is willing to provide them upon request.

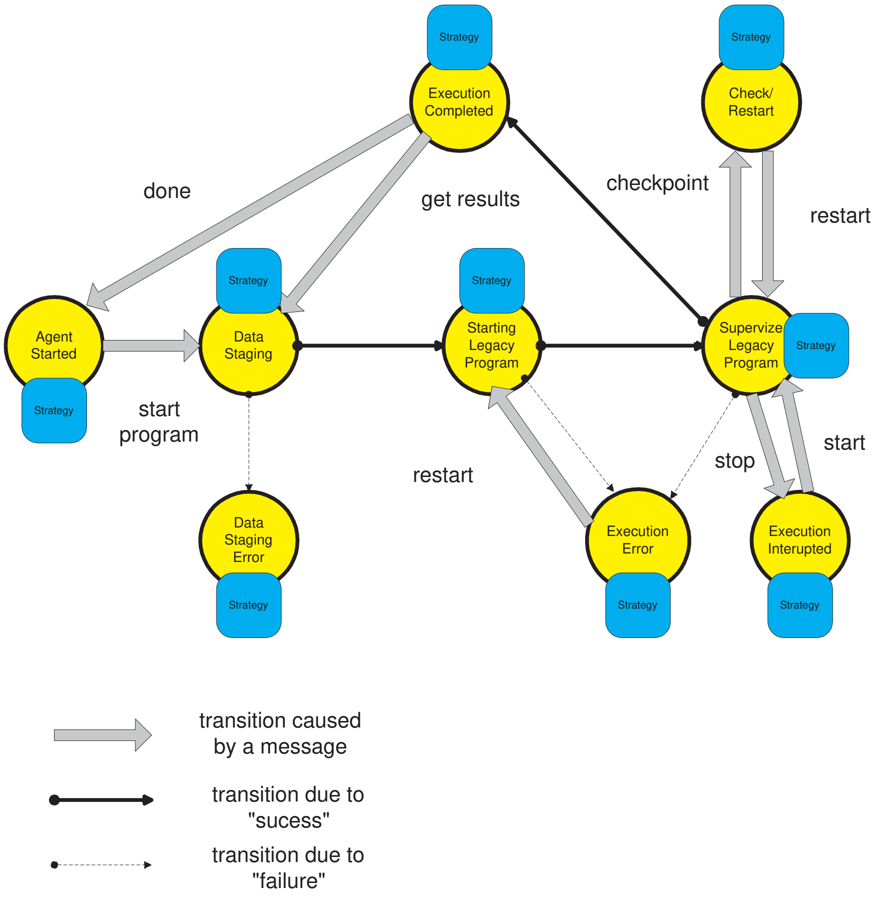

Figure 8: The state transition diagram of the Data Staging and Remote Execution Agent, `bondDSexec`. The functions of the agent are: perform data staging, start-up a legacy program on a remote host, and supervize its execution. State transitions are caused by: (a) external messages, (b) the success/failure to attain the goal of the strategy in a given state.

- *Monitor:* an objects that collects the properties of other objects, analyzes them, and takes actions based upon the result of this analysis.

In this model a monitor subscribes to the properties of interest of a subject. The subject notifies the subscribers of the events. The notifications are sent according to one of the notification models:

- *Time-triggered:* the subject notifies the changes periodically using an agreed-upon time interval.
- *Change-triggered:* any changes in the subscribed properties triggers notification.
- *Hybrid:* it works like change-triggered when changes do occur as time-triggered if there are no changes.

An implicit assumption of our model is that monitors and subjects are aware of the effects of communication latency and processing delays. If we denote by δ the total elapsed time from the instance of change, till the action triggered by the change is completed, the objects involved in monitoring maintain their relevant properties stable for at least Δ time with $\Delta > \delta$. We call this Δ*-monitoring*. Each pair (monitor, subject) needs to agree upon the value of Δ and considers only actions compatible with or irrelevant with this value.

A *Bond Monitoring Framework* implements this model. It consists of monitoring probe, monitoring interface, and monitoring policy. The *monitoring probe* is an object implementing the subscription/notification communication between a monitor and a subject. By attaching the monitoring probe, objects can perform that communication. The *monitoring interface* defines a list of methods which the objects with the monitoring probe should implement:

- `handleNotification()` is invoked whenever the notifications arrive.
- `handleMissingNotification()` is invoked when the expected notification is missing.
- `handleMissingProperty()` is invoked when the subject does not include the subscribed property.
- `handleError()` is invoked in the case of general errors.

The *monitoring policy* is an object containing a subject address, property names, and notification model. The monitor starts monitoring by creating the monitoring policy and handing it over to its monitoring probe.

Figure 9 shows a monitoring example. Suppose that there are two objects, A and B. A is going to monitor B. Both A and B create monitoring probes and implement monitoring interface. A constructs a monitoring policy and gives it to its monitoring probe, which in turn subscribes to B. The subscription request is delivered along the hierarchy to the monitoring probe of B, which sends back the notifications. The monitoring probe of A invokes `handleNotification()` whenever the notifications arrive.

5 Security Framework

Applications of network computing have vastly different security requirements and the trade-off between security and efficiency are application specific [12, 13].

When we consider the security in a distributed object middleware like Bond, it's very clear that a monolith design of security based on a particular security model is not appropriate. What we need is an extensible core security object that can support multiple security models

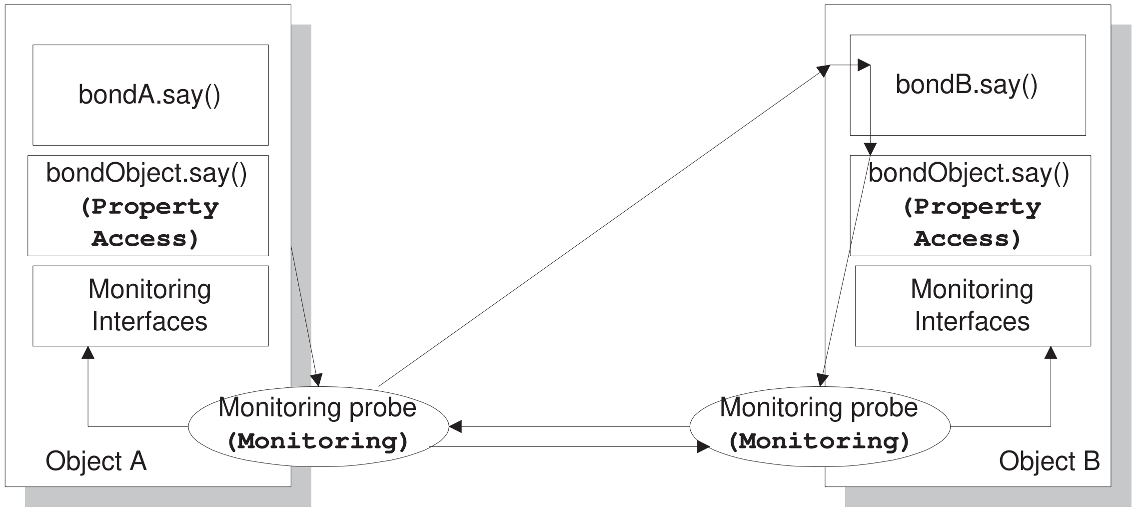

Figure 9: Two objects, A and B, involved in a monitoring relationship, A monitors B. Both objects create the monitoring probe and implement the monitoring interface. Once the monitoring is started, all the monitoring messages are exchanged between the monitoring probes.

and can be added to existing object dynamically. This requirement inspires the following design principles [8, 9, 10].

The first design principle of bond security is to provide a framework for security, not implementations based on any specific security models. By providing such a general framework, Bond leaves the decision of choosing security mechanism, such as the format of credentials, the authentication policy, the access control policy and etc. to system developer. Such a framework is implemented through an extensible core bond object called bondSecurityContext and a set of well-defined security interfaces.

The second design principle is that security is an aspect of a complex object design. A Bond object can become a secure object by declaring or dynamically adding the property bondSecurityContext. The object has this property is called 'master object'. The bondSecurityContext sets up a secure boundary for the object and intercepts all the messages sent to/from the object.

The third design principle is to 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. A bondSecurityContext contains security-related objects supporting security functions. Each of these security-related object implements one of the security interfaces defined by Bond.

The bondSecurityContext sets up a secure boundary for the master object. It has two methods: `incomingMessageProcess()` and `outgoingMessageProcess()`, called on the incoming and outgoing messages.

The bondSecurityContext is implemented as a preemptive probe to intercept all the incoming messages and outgoing messages. The bondSecurityContext captures all the messages sent to its Bond object, and the `incomingMessageProcess()` method invoked to enforce security measures without the involvement of the Bond object.

The bondSecurityContext captures all the messages sent out by the object. It achieves this by intercepting the message when it was moved from the object to the local message thread for delivering. The `outgoingMessageProcess()` method will be called to do any programmer defined security process on the outgoing message.

A BondSecurityContext contains security-related objects that implement various security

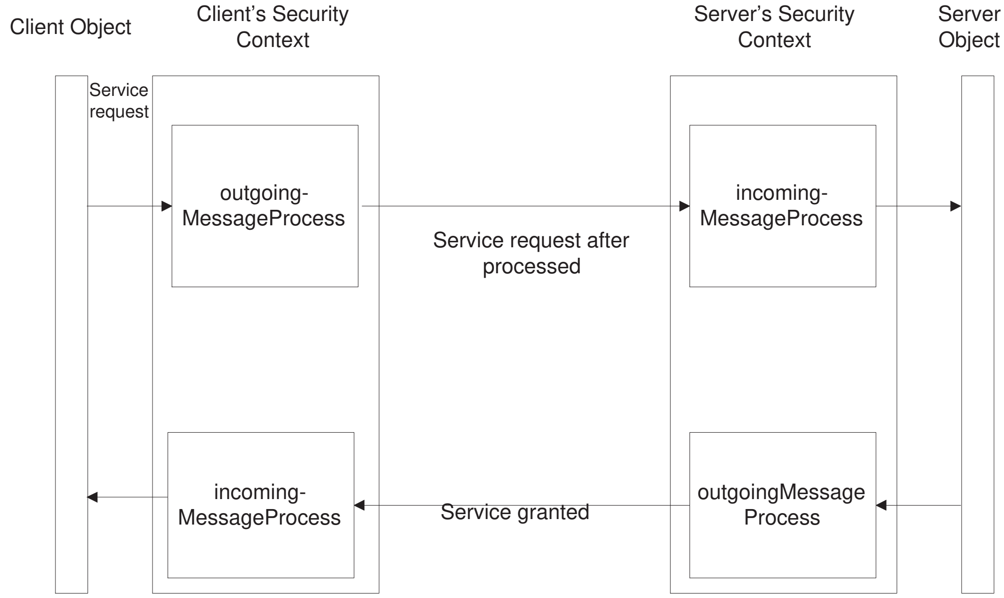

Figure 10: Application of the Bond security model to client-server operations. The client object first sends a service request to the server object. This message is intercepted by its security context, it is then forwarded to the `outgoingMessageProcess()` and sent out. All incoming messages are intercepted by the server's security context. The `incomingMessageProcess()` enforces authentication and access control before delivering the message to the server object. Finally, the server object grants the service.

functions. Each security-related object implements one of the security interface defined by Bond. Existing security interfaces includes:

- `bondCredentialInterface` - This interface defines the method to access the credential possessed by the current bondSecurityContext. The implementation of a credential can be user/password, access ticket or public/private key pair.

- `bondAuthenticatorInterface` - This interface defines the method used to enforce authentication for each received message. Programmer or administrator can define different authentication mechanisms, for example, username/password based authentication, ticket based authentication, or certificate based authentication. The only restriction is adhering to this interface. The only method in this interface is `authenticateClient()`, it will return an authenticated user identifier.

- `bondAccessControlInterface` - This interface defines the method to enforce access control for each received message. Programmer or administrator can define different access control mechanisms, for example, IP address based firewall, username based access control list, or others. The methods provided by this security interface are `initACL()` and `checkRight()`. The first method is used to initialize the access control object; and the second one is used to check access rights of an authenticated user identifier.

6 Bond Programing

First of all, Bond programs are regular Java programs. Regular Java programs can be extended to use some of the advanced features offered by the Bond objects. Programmers can use these advanced features either by using one of the object classes provided by Bond, or by deriving their own classes from a well chosen Bond object.

The Bond classes are in a jar file called `bond.jar`. Before starting, one have to make sure that this file is in the `CLASSPATH` variable (or it is specified in the list of libraries if you are using and integrated development environment like JBuilder).

Next, you have to import the bond needed Bond classes into the program. We recommand that every file should import at least the `bond.object` package:

```
import bond.object.*;
```

As we go forward, we may need to import some other packages also. These preparations being finished we may move forward to learn about the new features of Bond objects.

6.1 Property access, *set* and *get*

Let's assume that we have a class derived from bondObject, with a single integer field.

```
class mybondObject extends bondObject {
   int i;
}
```

Of course we know how to access the field `i`. Bond gives us a new way of performing the same task, with the function `get` which accesses the field it's name in string format. For example:

```
    mybondObject a,b;
    a.i = 7;
    System.out.println( a.i );
    System.out.println( a.get("i") );
```

Both print statements will print the value of `i`, 7. In the same way, the `set` function allows the program to set the a variable using it's name in string format:

```
    a.set("i", new Integer(8));
    System.out.println( a.i );  // prints 8
```

The same functions can be used to create new dynamic properties of the objects. Let's assume that we want to create a new member variable *j* of our object `a`. We can simply say:

```
    a.set("j", new Integer(10));
```

Of course we cannot access this variable using `a.j` but we can still use `a.get("j")` for the same effect.

As a conclusion, Bond objects have two types of member variables: regular Java *fields* and *dynamic properties* created when needed. The `set` and `get` methods allows access to both types of member variables.

Theoretically, one can then use only the basic `bondObject` type for every kind of data. In practice, access by name and dynamic properties are considerably slower than regular fields. A correct programming technique is to use dynamic properties for variables which cannot be anticipated as programming time (as we will see later, *probes* are a good example of this) while the `set` and `get` functions can be used for accessing dynamic properties or accessing objects remotely.

6.2 Messages, say

Bond objects have the intrinsic capability of sending and understanding KQML messages. Without entering into details of the KQML language, which can be consulted from [6, 7] we will present how messaging is done in Bond objects.

The `say` function is used send a message to a Bond Object. The function has two parameters: the message `String m` and the sender `bondObject sender`. If we are sending a message from a method of a Bond object, the sender is usually `this`. Let's assume now that we have two objects `a` and `b` derived from `bondObject` and being in a method of `b` we want to send a polite message to `a`. The code will be:

```
a.say("(tell :content hello :tone polite)", this);
```

One of the things we must be careful about, is that the message should be of valid KQML syntax. On the other end, in `b` we have the implementation of the say function as follows:

```
void say(bondKQML m, bondObject sender) {
   if (m.content.equals("hello")) {
      // give a reply
      sender.say("(tell :content howdy)", this);
      return;
   }
   super.say(m,sender);
}
```

The message already arrives in a parsed format, as a `bondKQML` object. The `content` is the only field of the `bondKQML object`. The parameters of the KQML message are dynamic variables of the object. This is one of those cases when we can not predict what kind of variables an object would have. So we can check the tone of the message using:

```
System.out.println( m.get(``:tone'') );
```

One would say that we didn't get too much by introducing messages, because `say()` is a local function call and we could do everything without bothering with messaging languages. The objects which gives legitimity to the `say()` function are the *shadows*. Shadows are the main communication abstraction of the Bond system, and the easiest way of thinking about them is that they are the local placeholder of a remote object. Let's assume that `a` and `b` reside on different hosts and `b` has a shadow of `a`. which is an object `shA` of type `bondShadow`. Now, `b` can send a message to `a` by:

```
shA.say("(tell :content hello :tone polite)", this);
```

We can see that the format is identical as before. The rule is that *messages sent to the shadow are messages sent to the object*. Another question is how the fact that the objects are now remote affects the implementation of the `say`. There is no need to write it here again, it is identical. Of course, the sender this time is not the original object, but a shadow of the original object.

What is happenning practically? The shadow object is a lightweight object which contains enough information to locate the remote object in its current networking environment. In the Internet world, this is the triplet of IP address, port and the Bond id of the remote object. When a shadow gets a message it contacts the *messaging thread* of the Bond program. The messaging thread delivers the message to the messaging thread of the remote program, which creates a shadow of the sender, looks up the destination object, and delivers the message to it as it would come from the sender's shadow.

One of the hidden, but important functions of the shadow is to *annotate* the message. KQML has two important parameters, `:sender` and `:receiver`. At the Bond programming level, we don't have to care about these parameters, because they are filled in by the shadow. If the message we sent was:

```
(tell :content hello :tone polite)
```

the message which will actually be transmitted will be:

```
  (tell :content hello
        :tone polite
        :sender B
:receiver A)
```

6.3 Locating Bond objects

Now it is time to move forward and think about how we can construct shadows to a Bond object. First of all if we know the *identifier* and the *location* of an object we can directly construct the shadow.

```
bondShadow bs = new bondShadow(``remote_id'', ``remote.machine.edu:2000'');
```

The identifier of the object is the value of the field `id` of the object. Every Bond object gets an unique identifier when created, and maintains it thoughout it's lifetime. As unique identifiers in general, it is long, meaningless for human users and aestetically ugly. The location of the remote object is the IP address and port the remote Bond program are registered. Bond programs typically are registering themselves to port addresses starting with 2000, and if that is not available, they try 2001, 2002 and so on.

Another way of creating a shadow for an object is to search for them in the directory. Every Bond program has a local directory where by default all Bond objects are registered (actually, you need to take special precautions not to be registered in the directory). The usage of the directory is very simple:

```
bondObject o = dir.find(``bondId'');
```

This call returns the object, if it is local to the program, a shadow to the object if it is remote and null if an object with a given identifier cannot be found. The way in which the directory can find a remote object is by contacting a *directory server*. If the network of directory servers is correctly set up, the above function call should find any possible Bond object on the internet, provided that there is a contiguous link of directory servers between them. The directory server also accepts as an input a *bondQuery* object.

6.4 Pairing of messages, `needReply()`, `onReply()`, and `replyWith()`

Usually message communication can be organizer in pairs (question-answer or message-acknowlegment) or sometimes in subscribe-unsubscribe relationships. If the communication is synchronous it is easy to keep track of the message pairs, because a question is immediately followed by a blocking wait for the answer.

Bond encourages the programmer to think in terms of asynchronous messaging. There are clear advantages to be gained: there is no blocking wait for the reply, more than one conversation can be handled at the same time, the program is more fault tolerant. However asynchronous communication is more difficult to program because the delay between a request and a reply can be arbitrary large and the sender may have changed its context before receiving the reply.

Bond provides the mechanism of the *waiting slots* to pair a late reply with the original request. The following exemplifies the use of this.

```
bondKQML m1 = new bondKQML("(ask-one :content invitation)");
m1.needReply();
shMary.say(m1, this);
bondKQML m2 = new bondKQML("(ask-one :content invitation)");
m2.needReply();
shKathy.say(m2, this);
```

So we send out two invitations without waiting the answer to any of them. The only difference is that we have marked them as messages which need reply - which for Bond means that it should create a *reply waiting slot* for the given question. On the other end, the invited person may reply as follows:

```
sender.say(m.replyWith(``(tell :content Yes)''));
```

The answer will be delivered ultimately in the waiting slot of the question. This means that instead the `say()` function of the object, the `on_reply()` function is called. If the object does not have an `on_reply()` function, the message will be delivered to `say()`.

```
int on_reply(bondKQML question, bondKQML reply) {
  if (reply.content.equals(``Yes'')) {
     if (question == m1) {
System.out.println(``Mary is coming!'');
     }
     if (question == m2) {
System.out.println(``Kathy is coming!'');
     }
  }
}
```

We can conclude, that the mechanism of reply waiting slots is automatically pairing an incoming reply with the question, even if there is an indefinite time between them.

6.5 Asynchronous communication in Bond: `waitReply()` and `ask()`

Although asynchronous communication has advantages over the synchronous, there are cases when we want explicitly to wait for a message. To continue the example of the previous section, if we want to wait for Mary's response before sending an invitation to Kathy, we should write:

```
bondKQML m1 = new bondKQML("(ask-one :content invitation)");
m1.needReply();
shMary.say(m1, this);
bondKQML reply = m1.waitReply(10000L);
```

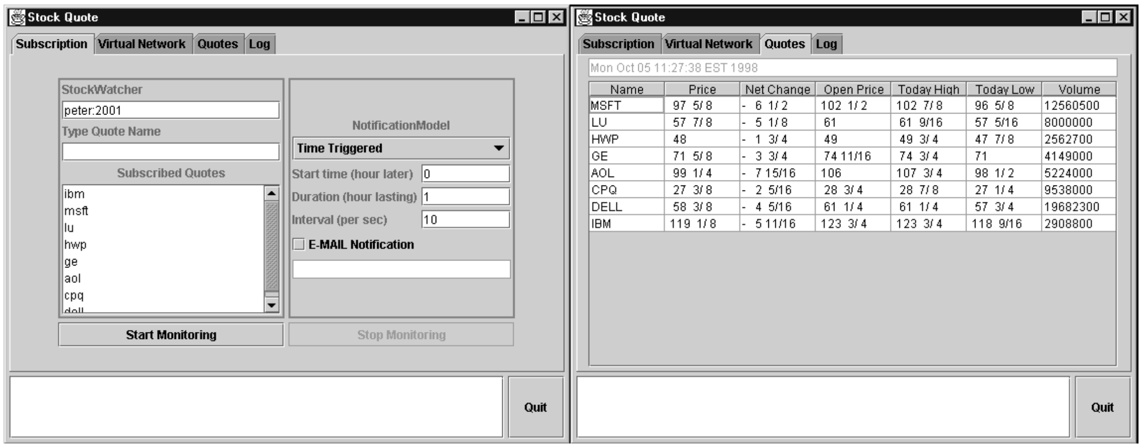

Figure 11: On the left, an interactive GUI for stock quotes subscription. Users specify quotes and notification model with related time parameters. On the right, the subscribed quotes are displayed.

The program will wait for Mary's reply to come or for the 10 second timeout (specified in milliseconds). If the timeout is reached, the function returns null, otherwise, returns the reply message. If there is no timeout specified, `waitReply` waits forever. The sequence above can be abreviated with the equivalent statement:

```
bondKQML reply = shMary.ask(``(ask-one content: invitation)'',this,10000L);
```

7 Demos

7.1 StockWatcher/Client: Data Source Monitoring using the Bond Monitoring Framework

The StockWatcher, SW, is a server providing stock quotes, each of which consists of a quote name, current price, trade volume, etc. The StockClient, SC, is an application allowing users to access the stock quotes of interest from the SW in real time. The communication between the SW and the SC conforms the subscription-based monitoring model, implemented using the Bond Monitoring Framework. A user subscribes to the SW by specifying a list of quote names and the notification model. Then the SW sends back the subscribed stock quotes, which in turn are displayed on the GUI of the SC, as shown in Figure 11.

Note that the Bond Monitoring Framework helps developers to concentrate on the application by providing monitoring functionality. Most effort for the SW was to develop quotes management, for the SC, it was the GUI implementation. The message exchanges for subscription and notification were easily implemented by the monitoring probes and the monitoring interface, even though that was the crucial part of this application.

References

[1] L. Bölöni: *Bond Objects – a white paper* Department of Computer Sciences, Purdue University CSD-TR #98-002

[2] L. Bölöni, K.K. Jun, T. Daniels and D.C. Marinescu: *Message patterns in the Bond Distributed Object System* Purdue University CSD-TR #98-004

[3] R. Orfali, D. Harkey, J. Edwards *Instant Corba* Wiley Computer Publishing 1996

[4] K. Mani Chandy, A. Rifkin, Paolo A.G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka and L. Weissman *A World-Wide Distributed System Using Java and the Internet* IEEE International Symposion on High Performance Distributed Computing, August 1996.

[5] K. Jun, L. Boloni, R. Hao, and D.C. Marinescu. A Subscription–based Monitoring Model for Distributed Object Systems. Submitted.

[6] T. Finin, et al. *Specification of the KQML Agent-Communication Language*, DARPA Knowledge Sharing Initiative draft, June 1993

[7] Y. Labrou, T. Finin *A Proposal for a new KQML Specification* UMBC TR-CS-97-03

[8] R. Hao, Bölöni, K. Jun, and D.C. Marinescu. *A Security Framework for Distributed Object middleware: Bond.* In preparation.

[9] R. Hao, Bölöni, K. Jun, and D.C. Marinescu. *Bond System Security And Access Control Model.* IASTED International Conference on Parallel and Distributed Computer and Networks, ParkRoyal Brisbane, Brinbane, Australia, December 14-16, 1998.

[10] R. Hao, Bölöni, K. Jun, and D.C. Marinescu. *Bond System Security And Access Control Model.* Technical Report, Department of Computer Science, Purdue University, CSD-TR#98-019,June 1998.

[11] B. Fairthorne. *OMG White Paper on Security.* OMG Security Working Group, April 1994.

[12] W.A. Wulf, C.X. Wang, D. Kienzle. *A New Model of Security for Distributed Systems.* Computer Science Technical Report CS-95-34, University of Virginia.

[13] J, B. Knudsen. *Java Cryptography.* O'Reilly, 1998.