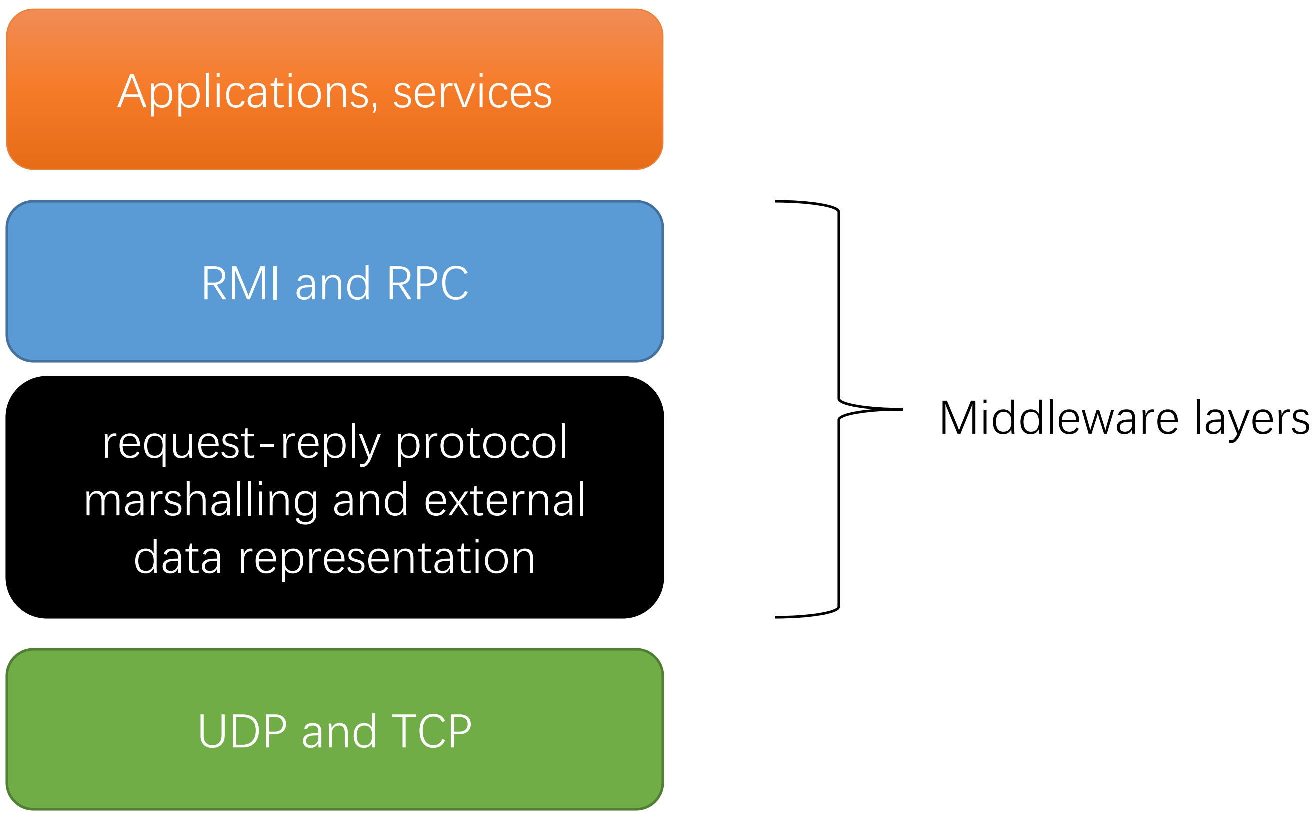
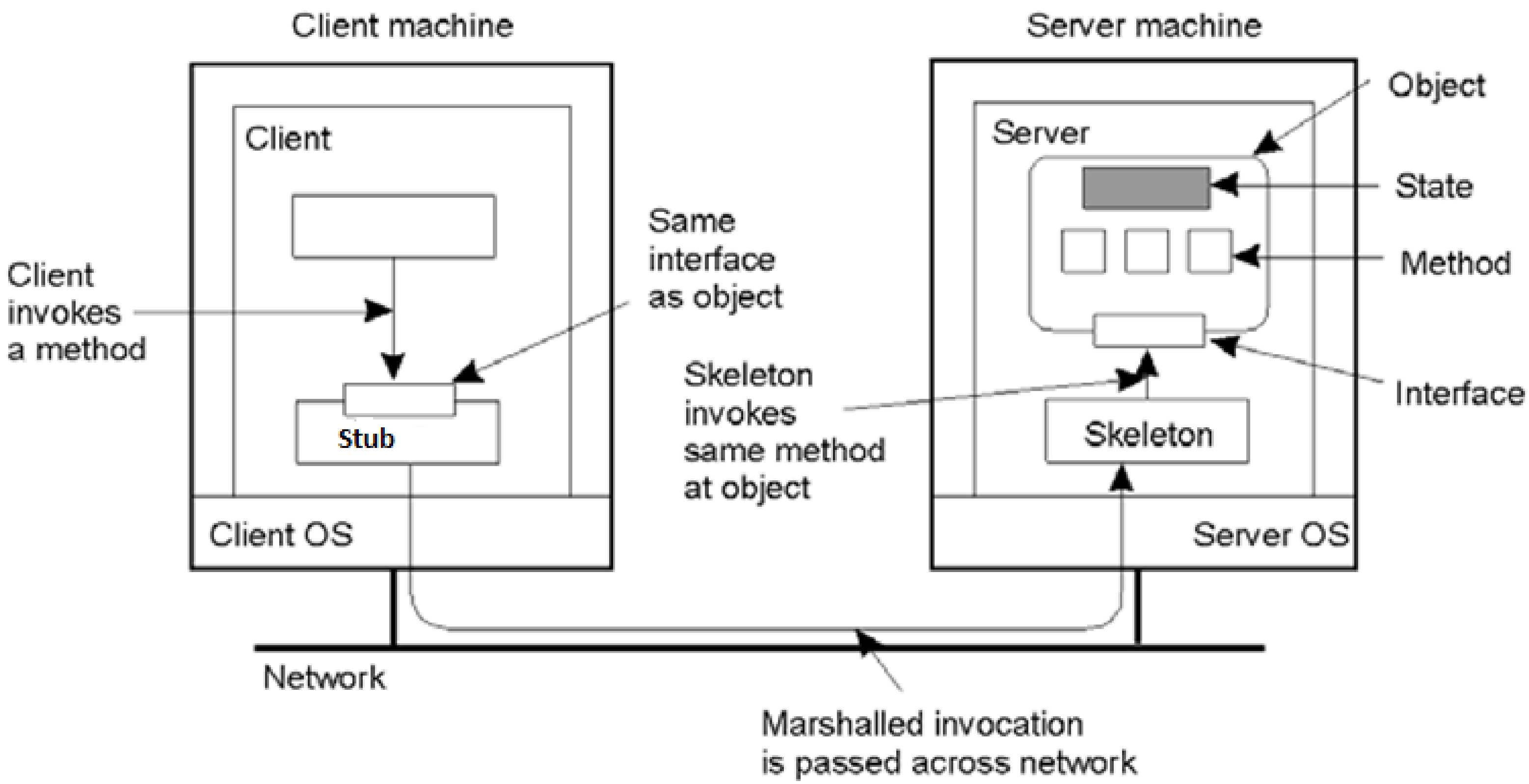
CPE 545

RMI (Remote Module Invocation)

Middleware layers



Distributed objects



Compile-time vs run-time objects

- Objects can be implemented in many different ways
  - Compile-time objects- Instances of classes written in object-oriented languages like Java, C++
    - Systems like Java RMI support compile-time objects
    - Not possible or difficult in language-independent RMI middleware such as CORBA
  - Run-time objects- Implementations of object interfaces are registered at an object adapter, which acts as an intermediary between the client and object implementation

Static vs dynamic RMI

- Static invocation
  - Typical way for writing code that uses RMI (similar to the process for writing RPC)
  - Declare the interface in IDL, compile the IDL file to generate client and server stubs, link them with client and server side code to generate the client and the server executables
  - Requires the object interface to be known when the client is being developed
- Dynamic invocation
  - The method invocation is composed at run-time invoke(object, method, input_parameters, output_parameters)
  - Useful for applications where object interfaces are discovered at run-time, e.g. object browser, batch processing systems for object invocations, “agents”

Design Issues for RMI

- Transparency
  - Should remote invocations be transparent to the programmer?
  - There are some differences between remote and local invocation:
    - Dealing with Partial failure, higher latency
    - Different semantics for remote objects, e.g. difficult to implement “cloning” in the same way for local and remote objects
  - Current consensus: remote invocations should be made transparent in the sense that syntax of a remote invocation is the same as the syntax of local invocation (access transparency) but programmers should be able to distinguish between remote and local objects by looking at their interfaces, e.g. in Java RMI, remote objects implement the Remote interface

Handling failures

- Types of failure
  - Client unable to locate server
  - Request message lost
  - Reply message lost
  - Server crashes after receiving a request
  - Client crashes after sending a request
- Invocation semantics
  - Maybe (no retransmission) , At-least-once (Re-execute procedure), At-most-once (Retransmit Reply)

Handling failures

Semantics	request retransmission (RT)	duplicate filtering (DF)	retransmission of results (RR)
Maybe	No	Not appl.	Not appl.
At-least-once	Yes	No	Re-execute procedure
At-most-once	Yes	Yes	Retransmit reply

Handling failures

- Server crashes
  - At least once (keep trying till server comes up again)
  - At most once (return immediately)
  - Exactly once impossible to achieve
- RPC
  - At least once semantics on successful call and maybe semantics if unsuccessful call
- CORBA, Java RMI
  - At most once semantics

Handling failures

- Lost request message
  - Retransmit a fixed number of times before throwing an exception
- Lost reply message
  - Client resubmits request
  - Server choices
    - Re-execute procedure → service should be idempotent so that it can be repeated safely
    - Filter duplicates → server should hold on to results until acknowledged

Handling failures

- Client crashes
  - If client crashes before RPC returns, we have an “orphan” computation at server
    - Wastes resources, could also start other computations
  - Orphan detection
    - Reincarnation (client broadcasts new “epoch” when it comes up again)
    - Expiration (RPC has fixed amount of time T to do work)

Distributed Garbage Collection

- Java approach based on reference counting
  - Server maintains a list of clients that hold remote object references for its remote objects
  - When a client first retrieves a remote reference to an object, it makes an addRef() invocation to server before creating a proxy (client reference)
  - When a clients local garbage collector notices that a proxy is no longer reachable, it makes a removeRef() invocation to the server before deleting the proxy
  - When the local garbage collector on the server notices that the list of client processes that have a remote reference to an object is empty (reference count is zero), it will delete the object

Distributed Binding

- Binding consists of two generic operations:
  - Export (server):
    - INPUT: Naming context and an object,
    - OUTPUT: A name for the object in the context.
  - Bind (client):
    - INPUT: Name of the object to be bound (object must have previously been exported).
    - OUTPUT: A handle that allows the object to be accessed.

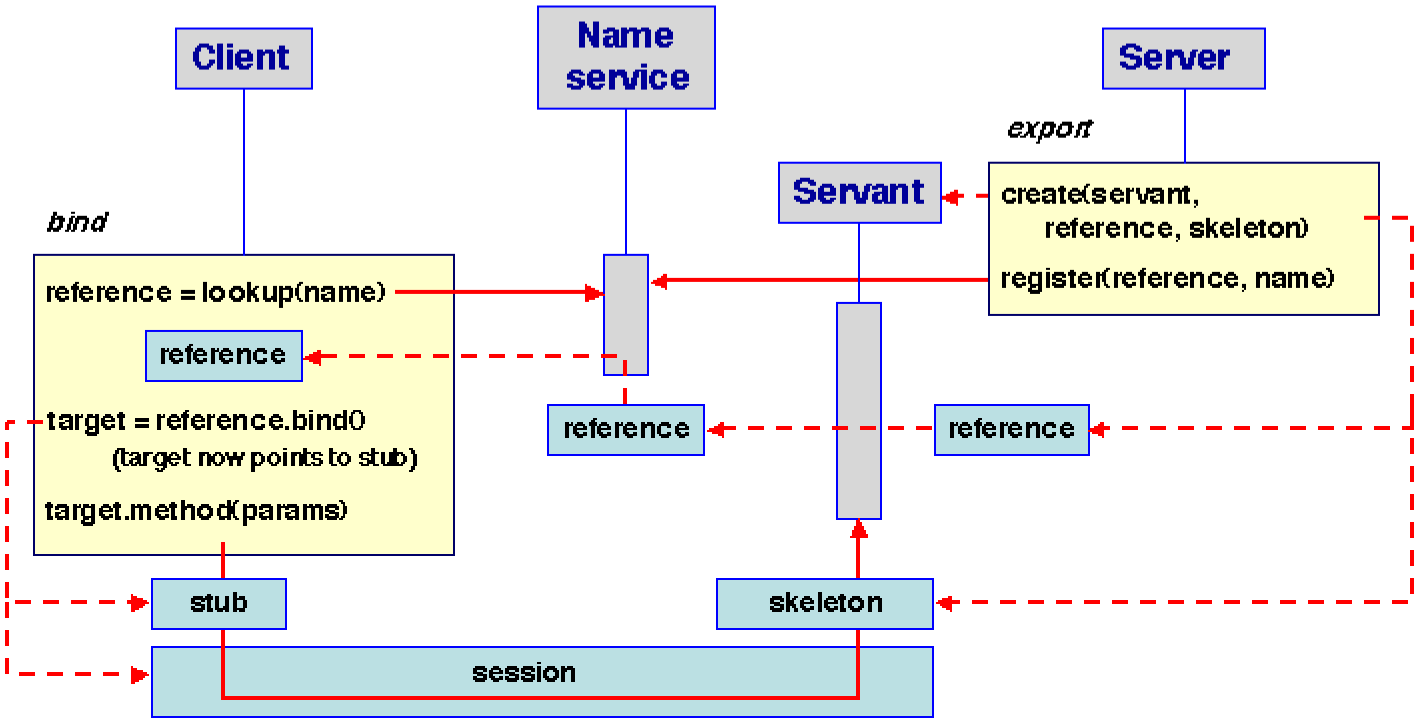
Binding in an ORB

- export:
  - Servant object is created (directly or through a factory)
  - Servant object is registered by the server (possibly using an adapter), thus providing a reference.
  - This reference is then registered in a name service.
  - Parts of the binding object (the skeleton and delegate instances) are also created at this time

- bind:
  - The client retrieves a reference for the servant, either through the name service
  - It uses this reference to generate an instance of the stub, and to set up the path from client to server by creating the end points of the communication path.

Binding in an ORB

•

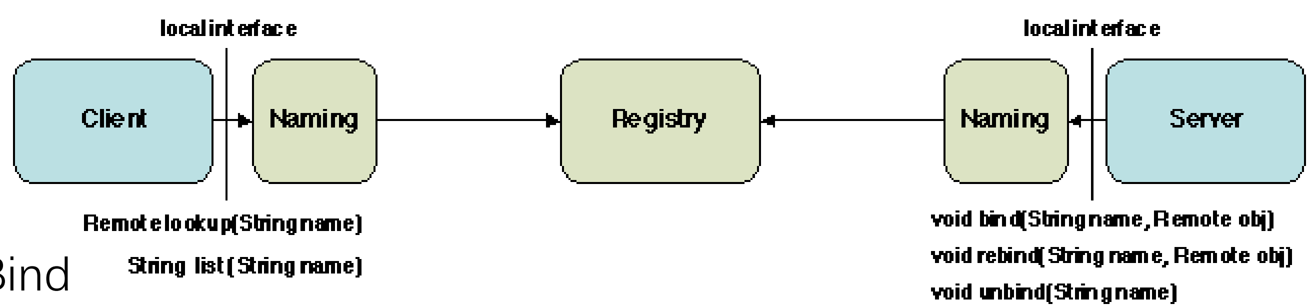


Java RMI

- A remote object system relies on a naming service. In Java RMI, this service is provided by a registry. The data that is registered is actually a reference for the remote object.
- The registry is accessible on both the client and server node through a local interface called *Naming*.

Java RMI

- The symbolic names have the URL format of the form:
  - **//[host name][:portname]/local name .**
  - **host is the host (remote or local) where the registry is located,**
  - **port is the port number on which the registry accepts calls,**
  - **name is a simple string uninterpreted by the registry.**
  - **Both host and port are optional. If host is omitted, the host defaults to the local host. If port is omitted, then the port defaults to 1099, the "well-known" port that RMI's registry, rmiregistry, uses**
- A server registers references in the registry using bind and rebind and unregisters them using unbind.
- The client uses lookup to search the registry for a reference of a given name.



Bind

Export

Java RMI

- Stub and skeleton classes are generated from a remote interface description, using a stub generator (rmi)

- Programming with remote objects is subject to a few rules:
  - Remote interface is defined like an ordinary Java interface, except that it must extend the *java.rmi.Remote* interface.

  - A call to a method of a remote object must throw the predefined exception *java.rmi.RemoteException*.

  - Any class implementing a remote object must create a stub and skeleton for each newly created instance; the stub is used as a reference for the object.
    - This is usually done by making the class extend the predefined class *java.rmi.server.UnicastRemoteObject*, provided by the RMI implementation.

Java RMI – (centralized)

```
// Hello Interface
     public interface Hello {
       String sayHello();}
     };
```

```
// Hello Implementation
    class HelloImpl implements Hello {
      HelloImpl() {   // constructor
        };
      public String sayHello() {
        return "Hello World!";
        };
    }
```

```
// Hello Usage
     ...
     Hello hello = new HelloImpl ();
     hello.sayHello();
```

Java RMI – Interface

```
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

// extends java.rmi.remote

// extends the predefined class java.rmi.server.UnicastRemoteObject
// object instantiated from this class is used to create stub and skeleton

```
class HelloImpl extends UnicastRemoteObject implements Hello {
    HelloImpl() throws RemoteException {
    };
  public String sayHello() throws RemoteException {
      return "Hello World!";
  };
```

Java RMI – Server

// The server creates the target object and registers it
// under a symbolic name (rebind operation).

```
public class Server {
  public static void main (...) {
    ...
    Naming.rebind("jrmi://" + registryHost + "/helloobj", new HelloImpl());
    System.out.println("Hello Server ready !");
  }
}
```

// Binds a url-formatted name (“jrmi://host/objectname”) to the remote object

Java RMI - Client

The client program looks up the symbolic name, and retrieves a stub for the target object, which allows it to perform the remote invocation.

stub

```
...
Hello obj = (Hello) Naming.lookup("jrmi://" + registryHost + "/helloobj");
System.out.println(obj.sayHello());
```

The client and server must agree on the symbolic name (how this agreement is achieved is not examined here).

Interface

```
import java.rmi.*;

 public interface Hello extends java.rmi.Remote {
    String sayHello() throws RemoteException;
}
```

Server

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject implements Hello
{
    public HelloImpl() throws RemoteException {}
    public String sayHello() { return "Hello world!"; }
    public static void main(String args[])
    {
        try
        {
            HelloImpl obj = new HelloImpl();
            // Bind this object instance to the name "HelloServer"
            Naming.rebind("HelloServer", obj);
        }
        catch (Exception e)
        {
            System.out.println("HelloImpl err: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

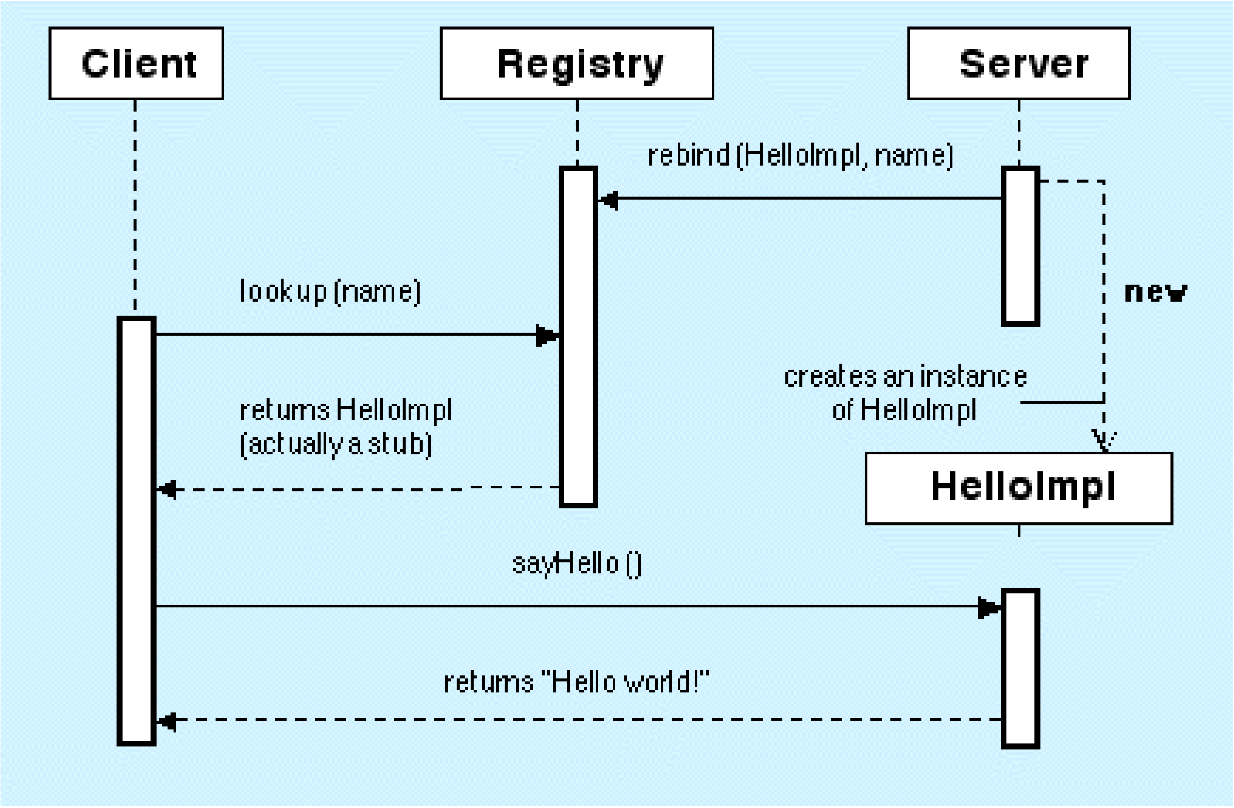
Client

```
import java.rmi.RMISecurityManager;
import java.rmi.Naming;
import java.rmi.RemoteException;
public class HelloClient
{
   public static void main(String arg[])
   {
      String message = "blank";
      // I download server's stubs ==> must set a SecurityManager
      System.setSecurityManager(new RMISecurityManager());
      try
      {
        Hello obj = (Hello) Naming.lookup( "//" +
            "lysander.cs.ucsb.edu" +
            "/HelloServer");        //objectname in registry
        System.out.println(obj.sayHello());
      }
      catch (Exception e)
      {
        System.out.println("HelloClient exception: " + e.getMessage());
        e.printStackTrace();
      }
   }
}
```

- Set the security manager, so that the client can download the stub code.
- Get a reference to the remote object implementation (advertised as "HelloServer") from the server host's rmiregistry.
- Invoke the remote sayHello method on the server's remote object
- *The constructed URL-string that is passed as a parameter to the* Naming.lookup *method must include the server's hostname.*

Java RMI

-


Client
Registry
Server
rebind (HelloImpl, name)
lookup (name)
new
creates an instance
of HelloImpl
returns HelloImpl
(actually a stub)
HelloImpl
sayHello ()
returns "Hello world!"

Agent Technologies

- Requirements for the next generation Telecom network:
  - Allow users to navigate through the massive content and service offerings
  - Allow for the potential of large bandwidth access to multimedia content and services
  - Allow for security mechanisms to be built in intelligent telecommunication infrastructures.
  - Allow for scalability through self-organization and hand-over between different networks

Agent Technologies

- The promise of agent technology in telecommunications is to provide:
  - Enabling more intelligence in service provision to allow value-added services and negotiation of QoS;
  - Dealing with the enlarging amount of information and functions, and allow self-organizing networks.

Agent Technologies

- Examples:
  - Software agent at a newspaper website can learn about a user’s preferences by tracking the user's actions, and then custom tailor the news summaries that suit the needs of that user.
  - Agents technology is used also to automate ordering process according to pre-determined inventory levels
  - Agents technology is used to search for cheaper airline tickets on the internet
  - A bidding agent can monitor online exchange activity for the best time to buy a commodity or material, and then inform the person via phone, text message, or email.

Agent Technologies

- An intelligent agent is “*a computer system, situated in some environment, that is capable of flexible, autonomous action in order to meet its design objectives.*
- Computer programs that use AI technology to 'learn,' and automate certain procedures and processes.

Agent Technologies

- Software agents are software components characterized by:
  - Autonomy - to act on their own
  - Reactiveness - to process external events
  - Proactiveness - to reach goals
  - Cooperation - to efficiently and effectively solve tasks
  - Adaptation - to learn by experience
  - Mobility - migration to new places

References

- Middleware Architecture with Patterns and Frameworks, Sacha Krakowiak
- Designing Embedded Communications Software, by T. Sridhar, ISBN: 157820125x, CMP Books
- IT Architectures and Middleware – 2nd edition. Chris Britton, Peter Bye. Addison-Wesley
- Tanenbaum & van Steen Distributed Systems: Principles and Paradigms, 2nd ed. ISBN: 0-132-39227-5. [Schmidt et al. 2000]Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. 666 pp.
- [Gamma et al. 1994]Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
- [Buschmann et al. 1995]Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1995). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons