Distributed Systems

8. Mobile Agents
Motivation for mobile computation (1)

- *Pervasive Networking and Computing*
  - Connectivity and computing is cheap and available

- *Ubiquitous Networking and Computing*
  - Connectivity and computing power is available everywhere (independently from location)

- *Mobile Computing*
  - Network nodes can be placed everywhere
  - Wireless communication

- *Easy-to-use Technologies for naive users*
  - World Wide Web
    - Electronic commerce
    - Internet phone
Motivation for mobile computation (2)

- Growing need for scalability
- Diffusion of network services and applications to large segments of the society
  - Customizability
  - Flexibility
  - Extensibility
- Well-established models and technologies
  - Essentially RPC-based
    - CORBA etc.
- Code mobility
  - The capability to dynamically change the bindings between code fragments and the location of execution
Code Mobility (1)

- A true distributed system is *location transparent*
- In a Mobile Computing System (MCS), applications are *location-aware*
  - Computational Environment (CE)
    - Location of the execution
  - Executions Units (EU)
    - Sequential flow of computation (e.g. a thread)
    - Code segment, data space, execution state
  - Resources
    - Sharable among EUs (e.g. files)
Code Mobility (2)
Strong mobility

- Code, data space and execution state of an EU migrate
- Migration
  - The EU is suspended, moved to the new CE and resumed
- Remote Cloning
  - Creates a copy of the running EU at the remote CE
  - The original EU is not detached
- Proactive migration or cloning
  - Time and destination are determined by the migrating EU
- Reactive migration or cloning
  - Time and destination are determined by another EU
- Strong mobility is entirely transparent to the user
- The transmitted code just resumes on the new CE
- Expensive
Weak mobility

• Code and data space can migrate, but *not* the execution state

• The migrated code will be restarted at a given procedure
  ➢ It starts in a similar way as an interrupt-handler

• Fetch or ship
  ➢ Either an EU fetches the code dynamically or ships code to execute to another CE

• Stand-alone code or code fragment
  ➢ Stand-alone code instantiates a new EU
  ➢ A code fragment is linked to an already running code

• Synchronous or asynchronous mobility
  ➢ Synchronous: requesting EU suspends execution
  ➢ Asynchronous: immediate or deferred
Data Space Management (1)

• Upon migration, bindings to resources must be rearranged

• Resource = (I, V, T)
  - I: unique identifier
  - V: value
  - T: type
  - Transferable vs. Not transferable
    - Principally not transferable is e.g. a printer
    - Transferable resources maybe
      - Free – can be freely migrate to another CE (e.g. a memory block)
      - Fixed – associated to a CE (e.g. a huge file)
Data Space Management (2)

- Resource binding to an EU may be
  - **By identifier**
    - The EU requires to be bound to a uniquely identified resource that cannot be substituted
    - E.g. a connection to a certain database
  - **By value**
    - The value (content) of the resource must not change due to the migration
    - The identity of the resource is not relevant
    - E.g. the value of a variable
  - **By type**
    - Bound resource must be compliant with a given type
    - Identity and value of the resource are not relevant
    - E.g. a system resource such as some network connection, or a block of memory
Data Space Management (3)

• Different bindings to the same resource is meaningful. E.g.
  ➢ An EU makes a binding by identifier to its display
  ➢ It makes a binding by type to the same resource
  ➢ After roaming it has actually two displays:
    ▪ The first one permanently associated to the original CE
    ▪ The second one at the actual CE, wherever it is

• Two classes of problems in data space management
  ➢ Resource relocation
  ➢ Binding reconfiguration

• Let be $U$ a migrating EU, with binding $B(\leftrightarrow)$ to resource $R$
Data Space Management (4)

- $U \leftrightarrow R$ by identifier
  - *Relocation by move*
    - Possible, if the resource is free transferable
    - May cause problems to other EUs with bindings to $R$
      - The other bindings might be removed
      - Or network references can be used to the new CE
  - *Network reference*
    - If the resource is fixed
    - May cause lot of network traffic

- $U \leftrightarrow R$ by value, $R$ is transferable
  - *Migration by copy*
  - A copy $R'$ of $R$ is created at the new CE
  - $B$ is changed on the new CE to a $B': U \leftrightarrow R'$
  - *Migration by move*
    - Possible, but generally inefficient
  - If $R$ is transferable but fixed
    - *Network reference* must be used
**Data Space Management (5)**

- **$U \leftrightarrow R$ by type**
  - **Re-binding**
    - B is voided at the old CE
    - B is re-established at the new CE to a new instance $R'$
    - No data transfer is necessary
    - A corresponding type and a free instance must exist!

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>By identifier</td>
<td>move (nw. ref.)</td>
<td>network ref.</td>
<td>network ref.</td>
</tr>
<tr>
<td>By value</td>
<td>copy (mo, nw. ref.)</td>
<td>copy (network ref.)</td>
<td>network ref.</td>
</tr>
<tr>
<td>By type</td>
<td>Re-binding (co, mo, n.r.)</td>
<td>Re-binding (co, n.r.)</td>
<td>Re-binding (network ref.)</td>
</tr>
</tbody>
</table>
Data Space Management (6)

By move

- Source CE
- Source CE
- Dest. CE

- Source CE
- Source CE
- Dest. CE

Nw. Ref.

- Source CE
- Source CE
- Dest. CE

Migrating obj.

R

voided

Network ref.

Network ref.
Data Space Management (7)

By copy

Source CE

Dest. CE

Copy value

Re-bind

Source CE

Dest. CE

Copy value
Design Paradigms (1)

• Components
  ➢ Code components – represents algorithms
  ➢ Resource components – data or devices
  ➢ Computational components – processors

• Interactions
  ➢ Events involving two or more components

• Sites
  ➢ Hosts computational components
  ➢ They represent the locations

• Interactions among components on the same site are cheaper than those located on different sites

• We assume that a computation can be carried out, if the following are all on the same site
  ➢ The know-how describing the computation
  ➢ The resources used during the computation
  ➢ The corresponding computational component
Design Paradigms (2)

• The computational component $A$ at site $S_a$ needs a service. There is a site $S_b$ that is involved in the service.

• Louise and Christine interact and cooperate to make a cake (result of the computation). They need:
  - A recipe (the know-how about the service)
  - The ingredients (movable resources)
  - An oven (a not moveable resource)
  - A person making the cake (computational component)

• The main design paradigms are:
  - Client/Server
  - Remote Evaluation
  - Code on Demand
  - Mobile Agents
Client-Server

- Louise would like to have a cake
- She does not know the recipe and she has neither ingredients nor an oven
- She calls Christine and asks her to make a cake for her
- Christine makes the cake and delivers it to Louise
- The client component $A$ at site $S_a$ sends a request to site $S_b$
- The server component $B$ at site $S_b$ performs the service using its own resources and returns the result to $A$
Remote Evaluation (REV)

- Louise wants to prepare the cake
- She knows the recipe but she has neither ingredients nor an oven
- She calls Christine, tells her the recipe and asks her to make such a cake for her
- Christine makes the cake and delivers it to Louise
- The client component $A$ at site $S_a$ sends the know-how (a procedure) to site $S_b$
- The server component $B$ at site $S_b$ performs this procedure using its own resources and returns the result to $A$. 
Code on Demand (CoD)

- Louise wants to prepare the cake
- She has both ingredients and oven, but she lacks the recipe
- She calls Christine, and asks her to tell the recipe
- Louise makes the cake
- The client comp. \( A \) at site \( S_a \) has the necessary resources but no procedure to process them
- It asks for the know-how at site \( S_b \)
- The server component \( B \) at site \( S_b \) delivers the know-how
- The client component \( A \) at site \( S_a \) executes this procedure
Mobile Agent (MA)

- Louise wants to prepare the cake
- She has both ingredients and the recipe, but no oven
- She prepares the cake
- She goes to Christine and completes the cake in her oven
- She eats it there or comes back with the cake
- The client component $A$ at site $S_a$ has the know-how
- Some of the required resources are at site $S_b$
- $A$ migrates to site $S_b$, carrying the know-how and maybe some intermediate results (some data) with
- $A$ completes the computation at site $S_b$, using its resources
- As opposed to REV and CoD, a whole computational component (incl. state and some resources) is moved
Selection of the paradigm

• There is no best paradigm nor a best technology
• Paradigm and technology are principally orthogonal
• In practice, they must conform to each other and application
• So, we have to clearly distinguish
  ➢ The application or application domain (e.g. a system to control a remote telescope)
  ➢ The design paradigm (e.g. Remote Evaluation)
  ➢ The technology used (e.g. Java Aglets)
• Mobile code applications are still rare
• Performance
  ➢ Mobile code is usually executed by an interpreter
Security issues

- If code can move easily, malicious code can this as well
- Authentication of the sender of mobile code
  - A server may want to authenticate the client
  - The client may also want to authenticate the server
- Which rights has the migrated code at the destination
- Sandbox
  - Dangerous calls are restricted by security control components
- Organizational approach
  - Allow mobile agents only to trustworthy institutions
  - Maybe to institutions with “good reputation”
- Manipulation detection
  - Does not protect again read attacks
- Blackbox
  - Obfuscate and invalid code before the attacker has time to crack it
Key benefits of mobile code (1)

• **Service customization**
  - Traditional Client/Server systems provide a fixed set of services – that is regularly upgraded by new versions
  - An alternative way is a simple server with dynamically extensible functionality provided by the client

• **Deployment and maintenance**
  - The new version of the software is installed on a server
  - Clients may load and dynamically link code fragments on demand – lazy propagation

• **Autonomous Components**
  - Communication channels have often low-bandwidth and low-reliability (e.g. wireless)
  - Constant connection is therefore not available (no C/S)
  - The user sends a single request to a stationary agent
  - The stationary agent makes the job, while e.g. the mobile equipment might be switched off
Key benefits of mobile code (2)

- **Fault tolerance**
  - In the client/server model the global state is distributed
  - Mobile agents just take their local state with themselves

- **Protocol Encapsulation**
  - Traditionally communication protocols must be installed at both peer entities
  - With mobile code one basic protocol suffices
  - Additionally, more sophisticated protocols can be downloaded with the help of the basic protocol

- **Software Engineering**
  - Mobile agents are well-suited to make rapid prototypes of a distributed system
  - Instead of a full installation, only a basic environment is installed
  - Prototyping code is distributed as mobile agents
Application Domains (1)

• **Distributed Information Retrieval**
  - Classical example for the MA paradigm
    - A mobile agent visits different hosts
    - It applies dynamic routing based on actual information
    - It performs search and filtering at the source of information, such decreasing network traffic

• **Active Documents**
  - Traditionally passive documents (as e-mail or Web pages) may be turned to be active by taking some code with
  - E.g. an e-mail takes the presentation software with it, thus allowing a presentation close to the original
  - E.g. an application that uses graphic forms to express queries to a remote database
    - The user requests the active document (CoD) and uses it as an interface (e.g with WWW+Java applets)
Application Domains (2)

• **Advanced Telecommunication Services**
  - Videoconference, video on demand, telemeeting etc.
  - They need dynamic reconfiguration and customization
  - E.g. the setup, signaling and presentation services could be dispatched to the users by a broker (REV)

• **Remote Device Control and Configuration**
  - E.g. network management
  - Configuration and monitoring code can be shipped (REV)

• **Workflow Management**
  - A mobile agent leads the workflow document through all stages of its processing in a distributed environment

• **Active Networks (CoD)**
  - E.g. routers can be dynamically “reprogrammed”

• **Electronic Commerce**
  - User transactions are carried out by a mobile agent
Mobile Agents

• Autonomous objects with behavior, state and location
  ➢ Can but need not to be “intelligent“ (in the sense of AI)

• Mobile Agent ↔ Service Agent Interaction
  ➢ Client/server like
  ➢ Ideally an RPC-like mechanism is provided

• Mobile Agent ↔ Mobile Agent Interaction
  ➢ Peer-to-peer, rather than client/server like
  ➢ Often not just request/response, general message passing

• Anonymous Agent Group Interaction
  ➢ The sender often does not know the receiver
    ▪ E.g. a group of agents is working on a problem
    ▪ The sender knows the group but not the individual agent
    ▪ Group communication can be used to this purpose
    ▪ Event channels can be used as well – additional level of indirection
<table>
<thead>
<tr>
<th>Languages</th>
<th>Mobility</th>
<th>Source</th>
<th>Avail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tacoma</td>
<td>Tcl, C, ...</td>
<td>REV</td>
<td>Uni Cornell</td>
</tr>
<tr>
<td>J. Servlets</td>
<td>Java</td>
<td>REV(push)</td>
<td>Sun</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COD (pull)</td>
<td></td>
</tr>
<tr>
<td>J. Applets</td>
<td>Java</td>
<td>COD</td>
<td>Sun</td>
</tr>
<tr>
<td>Active X</td>
<td>C, C++, ...?</td>
<td>COD</td>
<td>Microsoft</td>
</tr>
<tr>
<td>Aglets</td>
<td>Java</td>
<td>Weak migr</td>
<td>IBM</td>
</tr>
<tr>
<td>Mole</td>
<td>Java</td>
<td>Weak migr</td>
<td>Uni Stuttgart</td>
</tr>
<tr>
<td>Odyssey</td>
<td>Java</td>
<td>Weak migr</td>
<td>General Magic</td>
</tr>
<tr>
<td>Voyager</td>
<td>Java</td>
<td>Weak migr</td>
<td>ObjectSpac</td>
</tr>
<tr>
<td>AgentTcl</td>
<td>Tcl</td>
<td>Strong migr</td>
<td>Dartmouth</td>
</tr>
<tr>
<td>Ara</td>
<td>Tcl, C, Java</td>
<td>Strong migr</td>
<td>Uni Kaiserslaut</td>
</tr>
<tr>
<td>Jade</td>
<td>Java</td>
<td>Weak migr</td>
<td>TLAB, Torino</td>
</tr>
<tr>
<td>Emerald</td>
<td>Emerald</td>
<td>Strong migr</td>
<td>Uni Washington</td>
</tr>
</tbody>
</table>
Example MA, TicTacToe (1)

- The *TicTacToe* class is located statically at the client sites
  - It may emit *askToplay* calls, on user demand
- The server site accepts and answers these calls
  - The 1. player must wait until a second is coming
  - If the 2. request arrives, the server creates an agent and passes it to the 1. player (Player0), via the *goTo* method
- The players send the agent back and forth with the actual state of the game, via the *goToPlayer* method
- The agent starts running at the destination site at a method specified by the source site
- The selection of the method depends on state of the game (e.g. *printRemis* for undecided)
- If the game is over (one player *wins* resp. *remis*) they shut down the voyager daemon
Example MA, TicTacToe (2)

1) askToPlay

2) askToPlay

3) create agent

4) goTo: Moves Agent to Player 0

5) Agent plays “ping-pong ball”
public class TTTAgent implements ITTTAgent, Serializable {

    public void goTo() {
        // Called by the server. It sends the agent to player
        try {
            // Agent’s execution there starts at printFieldArrival
            Agent.of(this).moveTo(players[0], "printFieldArrival");
            . . .
        }

    }

    public void printFieldArrival() {
        . . . make own move … playTurn();
    }

    public void playTurn() {
        // Checks input and the effect of it
        . . .
        if (game not ended yet) {
            goToPlayer("printFieldArrival");
            else if (remis) { isRemis() }
            else if (winner) { isWinning() }
        }
    }
}
public void isWinning() //Print winning message
    {
        goToPlayer("printLooser");
        Voyager.shutdown();
    }

public void printLooser() //Print losing message
    {
        Agent.of(this).setAutonomous(false);
        Voyager.shutdown();
    }

public void goToPlayer(String callback)   {
    // Agent is moved – incl. instance variables (state of the game)!
    // The actual value of callback depends on the state of the game.
    // For normal case, printFieldArrival, for remis printRemis ...
    try {
        Agent.of(this).moveTo(players[actual++ % 2], callback);
        . . .    }
}
Example CoD, Server-Defined Policy (1)

- Server communicates with the clients via RMI and with a relational database via JDBC.
- Clients
  - Create expense records
  - Check the validity of the expense records
  - Send the proper reports to the server
- Server
  - Stores them in the database
  - The validation policy may change
  - The computation for validation the expense record can be offloaded from the server to the client
  - The policy for this check can be changed without any change in the client code
Example CoD, Server-Defined Policy (2)
Example CoD, Server-Defined Policy (3)

• The client may use the following interface remotely:
  
  public interface ExpenseServer extends Remote {
    Policy getPolicy() throws RemoteException;
    void submitReport(ExpenseReport report)
      throws RemoteException, InvalidReportException;
  } // ExpenseServer

• The policy interface itself is non-remote, a policy object will be copied to the client, who may use the checkValid method locally
  
  public interface Policy {
    void checkValid(ExpenseEntry entry)
      throws PolicyViolationException;
  } // Policy
A typical client looks like

```
Policy curPolicy = server.getPolicy();
// start a new expense report
// show the GUI to the user
while (user keeps adding entries) {
    try {
        curPolicy.checkValid(entry);
        // add the entry to the expense report
    } catch (PolicyViolationException e) {
        // show the error to the user
    } // try
} // while
server.submitReport(report);
```
Example CoD, Server-Defined Policy (5)

- Implementation of the server:

  class ExpenseServerImpl
    extends UnicastRemoteObject,
        implements ExpenseServer {
    ExpenseServerImpl() throws RemoteException { … }

    public Policy getPolicy() {
        return new TodaysPolicy();
    }

    public void submitReport(ExpenseReport report) { … }
} // ExpenseServerImpl
• The actual policy is defined by the following class

```java
public class TodaysPolicy implements Policy {
    public void checkValid(ExpenseEntry entry) throws PolicyViolationException {
        if (entry.dollars() < 20) {
            return; // no receipt required
        } else if (entry.haveReceipt() == false) {
            throw new PolicyViolationException;
        } // if
    } // checkValid
} // TodaysPolicy
```
Example CoD, Server-Defined Policy (7)

- **To change the policy**
  - Provide a new implementation of the interface `Policy`
  - Server returns `TomorrowsPolicy` instead of `TodaysPolicy` objects

```java
public class TomorrowsPolicy implements Policy {
    public void checkValid(ExpenseEntry entry)
        throws PolicyViolationException {
        if (entry.isMeal() && entry.dollars() < 20) {
            return; // no receipt required
        } else if (entry.haveReceipt() == false) {
            throw new PolicyViolationException;
        } // if
    } // checkValid
} // TomorrowsPolicy
```
Example REV, Compute Server (1)

- We want to delegate some computations to a special compute (e.g. very fast) server
- *Task* is a non-remote interface, with a *run* method that can be overridden by any computation. The signature of *run* is as "generic" as possible
  
  ```java
  public interface Task { Object run(); }
  ```

- The remote server takes a task from a client, executes it and returns the result.
public interface ComputeServer extends Remote {
    Object compute(Task task) throws RemoteException;
} // ComputeServer
public class ComputeServerImpl 
    extends UnicastRemoteObject, implements ComputeServer {
    public ComputeServerImpl() throws RemoteException { . . . }

    public Object compute(Task task) { return task.run(); }  // comp.+result
    
    public static void main(String[] args) throws Exception {
        System.setSecurityManager(new RMISecurityManager());
        ComputeServerImpl server = new ComputeServerImpl();
        Naming.rebind("ComputeServer", server);
        System.out.println("Ready to receive tasks");
        return;
    } // main
} // ComputeServerImp