Strong Code Mobility
Why Code Mobility?

- **Relocation** of services is necessary in environments where the context frequently changes.
- **Users moving about** geographically.
- Collaborating service components need to **migrate independently**.
- Migration must be **seamless**.
Work Package on Code Mobility

- Strong Mobility
- Progressive Mobility
- Smart Mobility

Proof-of-concept high-level virtual machine supporting strong code mobility

- High-level programs
- ChitChat VM
- Java Virtual Machine
Strong Mobility: Approach

Proof-of-concept high-level virtual machine
supporting strong code mobility

High-level programs
interpreted by
ChitChat VM
make run-time state explicit in Java objects
Java Virtual Machine
serialize state to bytestream
send stream through Java I/O sockets

High-level programs
resume execution
ChitChat VM
restore state
Java Virtual Machine
## Types of Mobility

<table>
<thead>
<tr>
<th>Mobility Type</th>
<th>Data Context</th>
<th>Control Context</th>
<th>Resources Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak Mobility</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Semi Strong Mobility</td>
<td>☑</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td><strong>Strong Mobility</strong></td>
<td>☑</td>
<td>☑</td>
<td>☒</td>
</tr>
<tr>
<td>Full Mobility</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

- Java Applets
- Most Middleware Solutions
- Threads ‘become’ java.io.Serializable
- Process Migration (for e.g. load-balancing)
Why Strong Code Mobility?

Semi strong mobility is far less expressive...

In an Aml environment, you cannot anticipate every move!

```java
public void m() {
    Object x = ...;
    o.n(this, x);
    // code hereafter never executed!
}

public void n(Object obj, Object x) {
    // ...
    move(obj, someLocation);
    obj.afterMove(x);
}

public void afterMove(Object x) {
    // perform some computation with x
}
```

should be executed at the remote location...
Why not just use ‘Java’?

- No provisions for mobility
- Middleware/language extensions
  - interfere with standard Java semantics
  - often give up JVM compatibility
- Technical problems with recursive transmission of classes
- A Virtual Machine can abstract from the underlying host system
Move Considered Harmful

- Imagine combinations of...
  - regular control flow (if, while, ...)
  - late binding polymorphism
  - meta-programming, reflection, aspects
  - move

- Which objects will be residing where?

  Conjecture: move is the ‘goto’ of mobility
  
  move (obj, 134.184.43.120) | goto 0xff408a7e
  move (doc, thePrinter)

  Wanted: Structured Mobility
  
  We need abstractions to control the “loci of objects”
ChitChat: Structured Mobility

- Model based on active objects
- New kind of method ‘modifier’: move
- Move methods ‘pull’ objects from one VM to another:

```o.m()```

```
move
```
Demo: Chat Client Application

- Simple client-server architecture
- Server automatically relocated to host of ‘most popular’ client

```java
sendMsg("Hello World")
```
Demo: Chat Client Application

**active object** `chatServer` {
    Object[] clients;
    int occupancy, maxClients, max;

    public `chatServer(channel, maxClients)` {
        clients = new Object[maxClients];
        occupancy = max = 0;
        this.maxClients = maxClients;
        this.register(channel);
    }

    public `move void come(nam)` {
        System.out.println("arrived at client "+nam);
    }

    public Object `registerClient(nam)` {
        **download** new `chatClient(nam)`;
    }
}

- a `chatServer` is an active object...
- ...with a reference to its connected clients
- move method used by clients to pull the server towards them
- weak mobility: client code can be downloaded by a remote machine
Demo: Chat Client Application

```java
active object chatClient extends chatServer {
    String nam; int count;

    public chatClient(nam) {
        this.nam = nam; count = 0;
        super {
            if (occupancy == maxClients)
                error("Sorry, channel is full");
            else
                clients[++occupancy] = this;
        }
    }

    public void receiveMsg(from, msg) {
        System.out.println(from + " ": "+msg);
    }

    public void sendMsg(msg) {
        if (++count > super.max) {
            super.come(nam);
            super.max = count;
            super {
                for (int i=0, i < occupancy, i++)
                    clients[i].receiveMsg(nam, msg);
            }
        }
    }
}
```

Clients have a dynamic relation with the server.

If I am the most active client, pull the server towards me.

Broadcast the message to all other clients.