Concurrent Object-Oriented Programming

Arnaud Bailly, Bertrand Meyer and Volkan Arslan
Lecture 1: Essential Definitions and Overview
Multi-Programming

- Only **one** physical node.
- A running program is called a **task** (sometimes process).
- Tasks are **totally independent** from one another.
- This defines the notion of “parallelism” (but **not** of parallel programming!).

![Diagram showing CPU, RAM, and tasks running on a single physical node.]
But there is an **Operating System**!
- The operating system occupies part of the memory.
- Tasks can execute OS code (through **system calls** or **interruptions**).
- The OS code may modify the OS memory.
- Tasks can thus **interfere** with each other.
- Several tasks (threads) created by some program.
- Primary objective: reduce the time needed for context-switching.
- “light” processes that share variables.
Parallel Programming (also!)

- At any moment, $n$ tasks are running! ($n > 1$)
- The memory will usually provide atomic r/w operations.
- There can be a global clock ("tightly coupled" systems).
- Different architectures (MIMD, SIMD) and memory consistency models...
- Different granularities (instruction, function, program).

![Diagram of Parallel Programming](image)
- Tasks can interfere **through the network**.
- Transmitted data is copied to/from the OS memory.
- The node is the **unit of failure**.
- **No global clock**.
- "**loosely coupled**" systems.
- Very different networks can be used.
Consider parallel programming or concurrent programming with preemptive scheduling.
Parallel read and write operations will be interleaved!
Possible executions:
\[
\{(x=2, y=2), (x=2, y=3), (x=3, y=2)\}
\]
Combinatorial explosion of the number of possible results when the number of parallel processes increases!
Reasoning is difficult.
• The notation is inspired by Hoare’s CSP.
• In $P$, the value of $x$ can be either 1 or 2.
• The choice is **nondeterministic**!
• One of the sending processes may never execute.
Some Additional Issues in

- Naming
- Heterogeneity
- Partial failures
- Load balancing
- Migration
- Security
- Scalability
Efficiency:
- multiplexing access to hardware resources,
- highly demanding computation (clusters),
- scalability and cost-efficiency.

Reliability through replication (back-up systems)

Necessity:
- Distributed nature of (embedded) systems,
- Providing services to distant clients.
Interaction paradigms for and

- Data-centric:
  - access to memory that is shared among tasks,
  - tasks have to synchronize to avoid interferences.

- Communication-centric:
  - no sharing of memory,
  - processes communicate by sending messages or calling routines (procedure, functions).

- Interference is avoided by:
  - client coordination (empty messages) or
  - supplier capacity to select incoming messages.

- Coordination-centric (tuple spaces as in Linda).
Avoiding interferences in and

- In the data-centric approach:
  - tasks have to **synchronize**.

- In the communication-centric approach:
  - client have to **coordinate** (by sending empty messages for example), or
  - the supplier has to select incoming messages.

- Coordination-centric: “loose” coordination.
Object-Oriented Programming

- **Object-based** programming provides:
  - encapsulation of data (information hiding),
  - well defined interface for operations (ADT),
  - identity.

- **Class-based** programming provides:
  - An abstraction and classification mechanism,
  - Code reuse through composition and inheritance.
  - Contracts (if you’re lucky).
They seem very different!
(even dealing with orthogonal concerns)

but

Robin Milner said:
"I can't understand why objects [of O-O languages] are not concurrent in the first place".
(Cited in [Matsuoka 1993].)
Why did Robin Milner say that?

- **Identifying** concepts:
  - **Object with task**, as
    - both (appear to) encapsulate data,
    - both have an autonomous behavior,
    - both are computational structures created at run-time.
  - **Routine invocation with message passing.**
With an after-look, this comparison seems rather deceptive, and overly simplifying.

- Variable sharing versus encapsulation?
- What about inheritance and composition?
- What about garbage collection?

Most of the O-O language mechanisms serve purposes that do concern neither [ ] nor [ ].
So... why should you follow this class?

- Concurrent and distributed programming is commonly done with objects.
- Many languages: almost any O-O language or platform has concurrent/distributed features.
- Ada 95, Java/RMI, Corba, C# and .NET...
Purpose of the Class

- Introduce the students to the diversity of O-O and O-O-O languages, those of today and those of tomorrow.
- Relate those languages to one another.
- Provide formal description tools to explore the foundational issues.
- Show how to reason about O-O and O-O-O (O-O) programs.
- Some parts will be quite practical, other more research-oriented.
Overview of Approaches and Platforms for and Programming
Distributed Applications

- Cluster/Grid computing (scientific computing)
- Ubiquitous computing (embedded systems)
- Client/server (the web)
- Peer-to-peer (large data exchange systems)
- Collaborating mobile agents (data retrieval)
- Others...
Possible Attitudes

Towards the following problems:
- Naming of Resources,
- Heterogeneity,
- Partial failures,
- Load balancing,
- Migration, and
- Security,

One can, when devising a programming language:
- Ignore the problem,
- Introduce primitives to address it, or
- Provide complete transparency (solve it).
Language approaches to ( ) and ( )

- The integrative approach
- The library approach
- The reflective approach

These approaches can be combined!

defined by [Briot et al. 98]
The Library Approach

- The most common way.
- Provides an API to the programmer.
- Wraps “native” code (e.g. system calls).
- Use through inheritance or composition.
- Approach of choice for middleware, agent platforms, messaging systems, etc.
The Integrative Approach

- **Identify concepts** found in the language with external ones.
- Introduce **new** (syntactic) constructions.
- It is the simplest approach.
- It leads to cleaner code.
- But it can’t address everything! (increase in language complexity)
- limitations: inheritance anomaly, etc.
- Difficult to modify a language (compilers, etc.)
Example: Java

- Possibility to create (concurrent) threads and to synchronize.
- Each object has an exclusive locking facility
- Creation of a thread by inheriting from `Thread`.
- `wait`, `notify`, `notifyall` are methods containing native code.
- `synchronized` is a key word.
- A method is identified with a monitor entry point.

```java
class Barrier {
    int num_waiting = 1

    // barrier
    public synchronized void join () {
        if (num_waiting < 3) {
            num_waiting += 1; wait
        } else notifyall ();
    }
}

class Client inherits Thread {
    Barrier b = new Barrier()

    public void run () {
        // joining the barrier
        b.join ();
        // all clients have joined
    }
}
```
Execution of the previous example

- Each thread has its own stack of calls.
- Objects do not belong to threads!
- Which thread is awoken by a notifyall is not specified
- Limited to (one CPU).

```
Thread 1  Thread 2  c1  c2  c3  b
Thread 3

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1.run</td>
<td>c3.run</td>
<td>c2.run</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b.join</td>
<td>b.join</td>
<td>b.join</td>
<td></td>
<td></td>
<td>b.join</td>
</tr>
<tr>
<td>b.wait</td>
<td>b.wait</td>
<td>b.notifyall</td>
<td>b.wait</td>
<td>b.notifyall</td>
<td></td>
</tr>
</tbody>
</table>

CPU Time
```
Threading primitives are clearly insufficient.

Example solutions relying on other libraries:

- Networking sockets,
- Java/RMI and Corba,
- Active objects and ProActive,
- Jini and JavaSpaces (based on Linda).
Java/RMI

- Provides a communication (RPC) layer.
- Compatible with Corba (IIOP in javax.rmi)
- Its interface:
  - a stub/skeleton generator (rmic)
  - a naming service (object registry)
Java/RMI step 1: Write an Interface

```java
import java.rmi.*;

public interface HelloInterface extends Remote {

/* return the message of the remote object, such as "Hello, world!". exception RemoteException if the remote invocation fails. */

    public String say() throws RemoteException;

}
```

- String is serializable (it can be marshaled)
import java.rmi.*;
import java.rmi.server.

class Hello extends UnicastRemoteObject implements HelloInterface {
    private String message;

    public Hello (String msg) throws RemoteException {
        message = msg;
    }
    public String say() throws RemoteException { return message; }
}

• Inherits from UnicastRemoteObject.
• “rmic Hello” will generate stub and skeleton.
• in main() method to register:
    Naming.rebind ("Hello", new Hello ("Hello, world!"));
Java/RMI step 3: Write a Client

```java
import java.rmi.*;

public static void main (String[] argv) {
    try {
        HelloInterface hello =
            (HelloInterface) Naming.lookup (“//se.inf.ethz/Hello”);
        System.out.println (hello.say());
    } catch (Exception e) {
        System.out.println ("HelloClient exception: " + e);
    }
}
```

- Uses the lookup function of the naming service.
- The remote object is accessed via a proxy (a.k.a. object handle, surrogate)
- type “rmiregistry” on the command line to start it.
The Reflective Approach

- Using reflection:
  - is equivalent to modifying the interpretation machine,
  - allows to go from program to data, and back (thunks).
- Certain languages (Scheme, Smalltalk) provide such capability.
- In O-O: use reflection to intercept method calls (reification).
- Re-routed may allow support for migration, replication, fault tolerance, etc. as a meta-program (MOP)
- It is often combined with the library approach.
- The code is often elegant.
- The execution is often inefficient.
- Orthogonal to re-usability.
Objects can be **active** or **passive**.

Communication is:

- **point-to-point**, and
- **asynchronous** (non-blocking send),
- respecting a **Call-by-value** policy.

Active objects have a **body**, that describe what messages they can receive (as in actors).

There are **No shared passive object**.
import java.net.*; import org.objectweb.proactive;

public class Hello /* DOES NOT INHERIT FROM ANYTHING */{
    private String name;
    // two constructors...
    public String sayHello() {
        return "Hello World"
    }
    public static void main(String[] args) {
        try {
            Hello hello = (Hello) ProActive.newActive(Hello.class.getName(),
                new Object[]{"remote"});

            InetAddress localhost = InetAddress.getLocalHost();

            ProActive.register(hello, "/" + localhost.getHostName()
                + "/Hello");
        } catch (Exception e) {/* error ... */
        }
    }
}
import java.net.*; import org.objectweb.proactive;

public class HelloClient {
    public static void main(String[] args) {
        Hello myServer; // !!!
        String message;
        try {
            if (args.length == 0) { // if there is no url to the server...
                }
            else {
                myServer = (Hello)ProActive.lookupActive
                           (Hello.class.getName(), args[0]);
                }
                message = myServer.sayHello();
                System.out.println("The message is : " + message);
            } catch (Exception e) { // error....
                } 
    }
}
The architecture uses RMI as underlying communication mechanism.

- The **Stub_B** reifies calls, **BodyProxy** deals with asynchrony, **Body** does the effective calls.
- **Almost** transparent to location (better than RMI).
- Supports migration.
- Limitations (final methods).
Jini and JavaSpaces

- Linda primitives:
  - objects read and write “tuples” into a shared space.
  - Communication is not directed.
  - write (t: Tuple) writes into the space.
  - Tuple read (p: Pattern) is blocking.
  - Tuple take (p: Pattern) is blocking and deleting.

- JavaSpaces adds:
  - non-blocking variants of the primitives,
  - transactions,
  - a leasing mechanism.
Other approaches to synchronization

- Group communication (multicast, broadcast).
- Message-Oriented Middleware (MOM).
- Component-based development (standardized interfaces such as COM or Enterprise Java Beans).
- All can be encoded using more elementary interaction patterns.
Conclusion

- Many ways to tackle the same problem!
- More features can be added.
- Many other criteria of classification.

To be continued...