Concurrent Object-Oriented Programming

Arnaud Bailly, Bertrand Meyer and Volkan Arslan

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).

Multi-Threaded Programming
- Several tasks (threads) created by some program.
- Primary objective: reduce the time needed for context-switching.
- "Light" processes that share variables.

Concurrent Programming
- But there is an Operating System!
  - The operating system occupies part of the memory.
  - Tasks can execute OS code (through system calls or interrruptions).
  - The OS code may modify the OS memory.
  - Tasks can thus interfere with each other.

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).
- Distributed Programming
  - Tasks can interfere through the network.
  - Transmitted data is copied as from the OS memory.
  - The node is the unit of failure.
  - No global clock.
  - "Loosely coupled" systems.
  - Very different networks can be used.

- Interference in
  - 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.

- Some Additional Issues in
  - Naming
  - Heterogeneity
  - Partial failures
  - Load balancing
  - Migration
  - Security
  - Scalability

- Interference in : Why bother?
  - Efficiency:
    - multiplexing access to hardware resources,
    - highly demanding computation (dusters),
    - 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 : Why bother?
  - 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).
  - Coordination-centric (tuple spaces as in Linda).
Avoiding interferences in $\mathbb{1}$ and $\mathbb{\triangle}$

- 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).

Blending O-O, $\mathbb{1}$ and $\mathbb{\triangle}$ (1)

- 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.**

But...

- 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 $\mathbb{1}$ nor $\mathbb{\triangle}$.

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 $\#$ and Δ 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 $\#$ and Δ (O-O) programs.
- Some parts will be quite practical, other more research-oriented.

Overview of Approaches and Platforms for $\#$ and Δ 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 Δ

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

These approaches can be combined!

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 = 0;
    public synchronized void join () {
        if (num_waiting == 0) {
            wait();
        }
        notifyAll();
    }
}
```

```java
class Client extends Thread {
    private Barrier barrier;
    public void run () {
        // Join the barrier
        // 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 awakened by a notifyAll is not specified.
- Limited to \( \mathbb{Z} \) (one CPU).

```
Thread 1:  c < e < b
Thread 2:  c < e < b
c < e < b
CPU Time:

c run

c run

c run

b wait

b wait

b notify

b notify

b notify

e run

e run

e run

c
```

The case for Java

- 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 java.rmi)
- Its interface:
  - a stub/skeleton generator (mic)
  - a naming service (object registry)

```
import java.rmi.*;
public interface HelloInterface extends Remote {
    String sayHello() throws RemoteException;
    String say() throws RemoteException;
}
```

Java/RMI step 1: Write an Interface

- String is serializable (it can be marshaled)
**Java/RMI step 2: Write a Server**

```java
import java.rmi.*;
import java.rmi.server.

class Hello extends UnicastRemoteObject implements HelloInterface {
    private String message;
    public Hello() throws RemoteException {
        super();
        message = "Hello world!");
    }
    public Hello(String msg) throws RemoteException {
        this.message = msg;
    }
    public String say() throws RemoteException { return message; }
}
```

- Inherits from UnicastRemoteObject.
- "rmic Hello" will generate stub and skeleton.
- in main() method to register:
  ```java
  Naming.rebind("Hello", new Hello("Hello, world!"));
  ```

**Java/RMI step 3: Write a Client**

```java
import java.rmi.*;

public static void main(String[] args) {
    try {
        HelloInterface hello = (HelloInterface) Naming.lookup("/se.inf.etna/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 (aka. 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 (thinks).
- Certain languages (scheme, Smalltalk) provide such capability.
- In O-O, use reflection to intercept method calls (reflection).
- Re-routed may allow support for migration, replication, fault tolerance, etc. as a meta-programs (MOOP).
- It is often combined with the library approach.
- The code is often elegant.
- The execution is often efficient.
- Orthogonal to re-usability.

---

**Example: ProActive**

- 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.

---

**ProActive step 1: Write a Server**

```java
import java.net.; import org.objectweb.proactive;

public class Hello { / DOES NOT INHERIT FROM ANYTHING */
    private String name;
    public Hello() { // two constructors...
        name = "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, localhostName() = "Hello");
        } catch (Exception e) { /* error ... }
    }
}
```

**ProActive step 2: Write a Client**

```java
import java.net.; import org.objectweb.proactive;

public class HelloClient { /
    public static void main(String[] args) {
        String message;
        try {
            if (args.length == 0) { // if there is no url to the server...
                message = "Hello, World!");
            } else {
                message = args[0];
            }
            Hello hello = (Hello) Proactive.createActive(Hello.class.getName(),
            new Object[]("Local");
            message = hello.say();
            System.out.println("HelloClient exception: "+ e);
    }
    ```

- Type "rmiregistry" on the command line to start it.
Architecture and Benefits

- 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...