Concurrent Object-Oriented Programming
Lectures Overview

- Lecture 1: Introduction and Motivation
- Lecture 2: Classic Approaches to Concurrent Programming
- Lecture 3: Objects and Concurrency
- Lecture 4: From Concurrent to Distributed Programming
Tasks can interfere through the network
Transmitted data is copied to/from the OS memory
No global clock
“Loosely coupled” systems
Very different networks can be used
Parallel computing can be done on distributed system
  - “Emulate” parallel hardware
  - Special case of distributed computing with strong assumptions

Is distributed computing vs concurrent computing just a matter of granularity?
  - Cf. threads vs tasks?
Lecture 4: From Concurrent to Distributed Programming
Outline

- Classic Approach

- Failure and system models

- Fundamental results

- Approaches to distributed (object-oriented) programming
“Classic” Approach

- Proxy/marshalling approach
  - Remote objects are invoked through proxies

- Two kinds of objects involved in (remote) interaction
  1. “Data objects” (small, likely primitive types)
     - Can be passed by value if arguments/return values to invocations
  2. “Bound objects” (larger, logically bound to host)
     - Are passed by reference, i.e., proxies are created on receiver site
Value Passing

Value Argument

Invoker  Proxy / Stub

I00II....I0I

Argument Copy

Skeleton  Invoker

Chair of Software Engineering

P. Eugster
Reference Passing

Invoker  Proxy / Stub  Reference Argument

Skeleton  Invokee  Argument Proxy

I00II...I0I
Passing Semantics

1. Pass-by-value (copy semantics)
   - Increases network traffic
   - No support for consistency/synchronization issues
   - Explicit distribution

2. Pass-by-reference
   - Increases dependency ("shared objects")
     - Failure-wise
     - Latency-wise
   - Might require heavy synchronization
   - Hidden distribution
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;

}

- Objects can/must be
  - Remote
  - Serializable
- Above, String is serializable
Interference in Distribution

- As long as no failures are considered
  - No additional ones
  - “Proxies” handled on behalf of remote objects
- But nothing is perfect
  - Failures can occur
    - Hosts
    - Tasks: usually unit of failure, but 1 per host
    - Communication
- Latency
  - A failed task (process) can not be distinguished from a very slow one
  - FLP-Impossibility result [Fischer,Lynch,Patterson’85]
Failures ↔ Exceptions?

- RemoteException is thrown in case of trouble
- But interpretation is often left to application
  - Has object received invocation?
    - Sent back return value?
  - Has object crashed?
    - In between request/return
- Resend request
  - What if received (in the meantime, return value lost)?
Proxy Proliferation

- Remote objects passed as arguments are transparently “replaced” by proxies

- Spiderweb of remote references is created
  - Increases dependencies between processes
  - A single failure/exception can bring down entire system
Imagine you are standing at an ATM withdrawing money...
Process View

ATM

withdraw(x)

[ok]

Bank/Account

account := account - x;
Latency ⇔ Asynchronous Invocations?

- Different variants
  - “Oneway”, i.e., no return value
    - Cf. actors
  - Lazy synchronization
    - A.k.a. wait-by-necessity (cf. SCOOP)
    - Object invoked, return value not “populated”
    - Implicit vs explicit future

- Issue
  - Asynchronously dealing with failures?
Unfortunately

- Latency and (partial) failures are reality!
  - Must cope with message passing latency, contention, ...
  - Failures of hosts/processes, transmission

- See [Gaertner’99] for a good overview

- Distributed systems are
  - Heterogeneous (without mentioning languages)
  - Not static
  - ...
  - Basically just a set of processes \( \{ p_1, p_2, \ldots \} \)
    communicating by message passing
Processes

- One assumes processes
  - Have unique identifiers
  - Are connected pair-wise through links which enable message passing

- At a tick of its local clock, a process
  - Executes a computation (local event) and exchanges messages with others (global event)
  - Note: one message sent/delivered per tick

- For a given algorithm run, a process is **correct** or **faulty**
### Failure Models

- **Omission**
  - Certain actions can be omitted, e.g., message reception or send (e.g., due to buffer limitations)

- **Crash**
  - (Usually) processes, e.g., crash-stop, crash-recovery

- **Timing**
  - Certain action takes place too late (usually based on assumption that this can be detected, i.e., bound)

- **Byzantine** (arbitrary)
  - Malicious behavior
(Message Passing) System Models

- **Synchronous**: known upper bounds on
  - *delays* for message transfer
  - *processing* speed
  - *drifts* between individual local host clocks and global real time

- **Asynchronous**: none of the above
  - Yet typical of Internet

- Intermediate
  - Partially synchronous (e.g., “failure detector”)
  - Eventually synchronous
Communication Channels

- Latency is one thing
  - If at least a message is “eventually” received

- **Unreliable** channels ("at-most-once")
  - *No duplication*: a message is received at most once
  - *No creation*: a message received has been sent

- **Fair-lossy** channels
  - *Fairness*: a repeatedly sent message is only lost a finite number of times (processes are correct)
  - Sufficient to build **reliable** channels: message is eventually received (processes are correct)
In Practice?

- UDP: unreliable channel

- TCP: reliable, and FIFO, assuming
  - Unlimited buffers
  - Unique identifiers for
    - Messages
    - Processes

- “Eventually”, and hence latency, are usually defined in terms of an algorithm run
  - Inversely run is bound by actions happening “eventually”..
Further

- **Consensus** impossible in asynchronous system [Fischer, Lynch, Patterson’85]
  - With a single process failure
  - Even with reliable channels

- Etc.
  - *Mutual exclusion* among \( n \) processes can not be implemented in asynchronous system with process failures
  - *(Non-blocking)* atomic commit
  - Leader election
  - ...
Imagine a process $p$ entering a critical section
- Process $p$ crashes before exiting
- Process $p$’s failure must be detected such that someone else can enter the critical section
- To that end, every process must periodically “hear” from every process (failure detector)
- When can a process which does not hear from $p$ decide that $p$ is dead?
  - No bound on communication latency: $p$ could still be alive (false suspicion)
  - Also, other processes might still have heard from $p$
In other Terms

- For reasoning about programs
  - We need to reason about (certain) operations
  - Yet, operations are usually divisible

- In the face of concurrency
  - Can introduce indivisible operations, e.g.,
  - masking interruptions in `lock()` and `unlock()`

- In the face of failures
  - By making sure that an action takes full effect or none
  - But how? Failures are not controlled by us.
Example: Reliable Broadcast

- **Integrity**: a message is \( r\text{delivered} \) at most once, and only if it was previously \( r\text{broadcast} \)

- **Validity**: if a correct process \( p \) \( r\text{broadcasts} \) a message \( m \), it eventually \( r\text{delivers} \) \( m \)

- **Agreement**: every message \( r\text{delivered} \) by a correct process \( p \) is \( r\text{delivered} \) by every correct process \( q \)

With reliable channels \((r\text{send} / r\text{receive})\):

\[
\text{broadcast}(m): r\text{send}(m) \text{ to every process } p
\]

upon \( r\text{receive}(m) \) for the first time:

\[
\text{r\text{send}(m) to every process } p
\]

\[
\text{deliver}(m)
\]
Attitudes towards Failures

- **Note**
  - Assumptions are necessary given impossibility results

- **Agnosticism**
  - Safety is mainly issue of software, i.e., data (type) safety
  - Strong assumptions, e.g.,
    - Reliable channels, no process failures
  - Static analysis based on static system, e.g., set of processes, set of exchanged data/messages
Attitudes (cont’d)

- **Awareness**
  - **Failure handling**
    - Make failures explicit, e.g., exceptions
    - Force programmer to deal with them
  - **Failure masking**
    - Employ specific protocols (based of course on additional assumptions on system)
    - Typically obtained by replication

- **Goal**
  - Maximize simplicity for programmer (abstraction)
  - Minimize assumptions and complexities of mechanisms
Typical Assumptions/Workarounds

- Often, reliable FIFO channels

- Oracles, e.g.,
  - **Failure detectors**
    - *Accuracy*: measure of false suspicions
    - *Completeness*: measure of righteous suspicions

- Randomization

- Majority of **correct** processes
Flavors of Awareness

- Failure handling
  + Usually rather lightweight mechanisms
  - Sometimes huge burden for programmer
  + Sometimes desired

- Failure masking
  - Usually rather heavyweight mechanisms
  + Relieves programmer from any burden
  + Sometimes necessary
Classic Mechanisms

- **Replication** (masking)
  - *High-availability* (crash only decreases performance slightly)
  - Heavyweight

- **Transactions** (handling)
  - Make *reasoning* in case of partial failures easier
  - Supports (heroic) efforts of application in such cases

- **Persistence**
  - Low-availability (crash affects performance widely)
  - *Lightweight*
Issues

- Consistency
  - All replicas must see commands/queries in same order
    - Total Order Broadcast et al.
- Nesting
  - What if replicas invoke further (replicated objects)
    - $N$-to-$M$ invocations
- Transparency
  - What to do with multiple return values
  - How to implement state transfers?
(Total) Order
Nesting
State Transfer
public void transfer(BankAccount account1,
                   BankAccount account2) {
    Transaction t = new Transaction();
    try {
        t.start();
        my1stAccount.withdraw(amount);
        my2ndAccount.deposit(amount);
    } catch(…) { … t.abort();… } 
    } finally { t.commit; }
}

or

public void transfer(BankAccount account1,
                   BankAccount account2) {
    atomic {
        my1stAccount.withdraw(amount);
        my2ndAccount.deposit(amount);
    } catch(…) { … }
Issues

- Consistency
  - **Serialization** of operations
  - Reducing locking (optimisitic)
- Nesting
  - Dealing with long transactions
- Transparency
  - Propagating transactional contexts
  - Expressing transactions in language
    - Inheritance et al.
  - Rollbacks/compensation (optimistic transactions)
Serialization Order

T1

myAccount.close()
myAccount.withdraw()

T2

myAccount.withdraw()
myAccount.balance()
Context Propagation
Approaches to and

- Possibilities [Briot et al. ‘98]
  1. The library approach
  2. The reflection approach
  3. The integration approach
Library Approach

- Group communication toolkits
  - Isis [Birman et al.], ...
  - In Java JGroups etc.

- Explicit way of going about replication
  - Programmer deals with
    - Plurality (multiple replies)
    - State transfer: implement `get-/setState()`
    - Joining, leaving
    - Required guarantees: reliable, total order, causal order, ...
public class MyServer extends Skeleton implements Replica {

    public Object getState() { ... }
    public void setState(Object state) { ... }

    public MyServer() { super(); join("/myGroup"); }

    public Object receive(Object[] args) { ... }
    ...
}

public class MyClient {

    ...

    public static void main(String[] args) {
        Group g = Group.lookup("/myGroup");
        g.setProtocol("tobcast");
        Object[] rets = g.broadcast(...);
        ...
    }
    ...
}
Transactions

- Transactions can be similarly dealt with

```java
public class MyTransactionalServer ... implements Transactional {
    public boolean vote() { ... }
    public void commit(...) { ... }
    public void abort(...) { ... }
    ...
}
```

- And even

```java
public class MyTransactionalServer ... implements Transactional {
    public void myMethod1(myArg1 arg1, ..., TransactionContext ctxt) {...}
    ...
}
```
Evaluation

- No surprises

- Servers/replicas must implement API
  - For state transfer, commit etc.

- Common approach: application server
  - Components are replicated
  - Run under the control of container
  - Every in-/outgoing invocation is intercepted
Reflection Approach

- Initial idea based on **Interceptors**
  - Filters for in-/outgoing invocations

- Flown into **Aspect-Oriented Programming**
  - Replication aspect
  - Transaction aspect
  - ...
Example

```java
public class Teller {
    public void transfer(BankAccount account1,
                          BankAccount account2) {
        my1stAccount.withdraw(amount);
        my2ndAccount.deposit(amount);
    }
}

aspect Transactionizer {
    pointcut transfer() :
        calls(void Teller.transfer(BankAccount, BankAccount));
    around() : transfer() {
        boolean aborted = false;
        Transaction t = new Transaction();
        try { proceed(); } finally {
            if (!aborted) t.commit();
        }
    }
}
```
Evaluation

- Known benefits of AOP/reflection approach
  - Separation of concerns
    - Divide efforts and expertise among programmers

- Known disadvantages
  - Transparency in main code misleading
    - Falsely promotes orthogonality
Integration Approach

- **Transactions**
  - Need to denote “atomic” sections (e.g., methods)
  - What to do in case of abort?
    - Client-side: retry automatically or exception
    - Server-side: automatic rollback?

- **Replication**
  - Replicated objects are invoked as usual through proxies
  - Multiple replies and policies handled through libraries
Example

```plaintext
... transfer (a1, a2: separate ACCOUNT; amount: INTEGER) is
   require
    a1.balance => amount
   atomic
    a1.withdraw(amount)
    a2.deposit(amount)
   ensure
    a1.balance = old a1.balance - amount
    a2.balance = old a2.balance + amount
   end

keep_trying is
do
   transfer(bill_gates_account, my_account, 1000000)
rescue
   retry
end
...
```
Evaluation

- Clearly cleanest approach
  - Division of responsibility
    - without false promises

- Drawback: interoperability
  - Not all languages support it
    - Somebody has to lead the path...