7066 Concurrent Programming

Prof. O. Nierstrasz

Wintersemester 1997/98

Table of Contents

1. Concurrent Programming

Lecturer:	Prof. O. Nierstrasz Schützenmattstr. 14/103, Tel. 631.4618
Secr.:	Frau I. Huber, Tel. 631.4692
Assistants:	M. Lumpe, F. Achermann, J.C. Cruz, S. Tichelaar
WWW:	`http://iamwww.unibe.ch/~scg/Lectures/`

Text:

❑ D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996

Other Sources:

❑ D. Lea, *Online Supplement to Concurrent Programming in Java,* `http://gee.cs.oswego.edu/dl/cpj/index.html`

❑ N. Carriero and D. Gelernter, *How to Write Parallel Programs: a First Course*, MIT Press, Cambridge, 1990.

❑ A. Burns and G. Davies, *Concurrent Programming*, Addison-Wesley, 1993

Schedule

❑	10.24	1. Introduction — Concurrency and Java	
❑	10.31	2. Safety	
❑	11.07	3. State-dependent Action	
❑	11.14	4. Asynchronous Methods	
❑	11.21	5. Fine-grained Synchronization	
❑	11.28	6. Architectural Styles for Concurrency	
❑	12.05	7. Coordination Languages	— *Juan Carlos Cruz*
❑	12.12	8. Coordination Components in Java	— *Sander Tichelaar*
❑	12.19	*No lecture*	
❑	01.09	9. Object-Based Concurrency	— *Juan Carlos Cruz*
❑	01.16	10. Petri Nets	
❑	01.23	11. Pi Calculus and Pict	— *Markus Lumpe*
❑	01.30	12. JPict — Pict in Java	— *Franz Achermann*
❑	02.06	*Open ...*	

Overview

- ❑ Concurrency and Parallelism
- ❑ Applications of Concurrency
- ❑ Limitations
  - ☞ safety, liveness, non-determinism ...
- ❑ Approach
  - ☞ idioms, patterns and architectural styles
- ❑ Java and concurrency

Concurrency and Parallelism

"A *sequential program* specifies sequential execution of a list of statements; its execution is called a *process*. A *concurrent program* specifies two or more sequential programs that may be executed concurrently as *parallel processes*."

A concurrent program can be executed by:

1.	*Multiprogramming:*	processes share one or more processors
2.	*Multiprocessing*:	each process runs on its own processor but with shared memory
3.	*Distributed processing:*	each process runs on its own processor connected by a network to others

Assume only that all processes make positive finite progress.

Applications of Concurrency

There are many good reasons to build concurrent programs:

- ❑ Reactive programming
  - ☞ minimize response delay; maximize throughput
- ❑ Real-time programming
  - ☞ process control applications
- ❑ Simulation
  - ☞ modelling real-world concurrency
- ❑ Parallelism
  - ☞ exploit multiple CPUs for number-crunching; exploit parallel algorithms
- ❑ Distribution
  - ☞ coordinate distributed services

Limitations

But concurrent applications introduce complexity:

- ❑ Safety
  - ☞ synchronization mechanisms are needed to maintain consistency
- ❑ Liveness
  - ☞ special techniques may be needed to guarantee progress
- ❑ Non-determinism
  - ☞ debugging is harder because results may depend on “race conditions”
- ❑ Communication complexity
  - ☞ communicating with a thread is more complex than a method call
- ❑ Run-time overhead
  - ☞ thread construction, context switching and synchronization take time

Atomicity

Programs P1 and P2 execute concurrently:

	{ x = 0 }
P1:	x := x+1
P2:	x := x+2
	{ x = ? }

What are possible values of x after P1 and P2 complete?

What is the *intended* final value of x?

Synchronization mechanisms are needed to restrict the possible interleavings of processes so that sets of actions can be seen as atomic.

Mutual exclusion ensures that statements within a *critical section* are treated atomically.

Safety and Liveness

There are two principal difficulties in implementing concurrent programs:

Safety — ensuring consistency:

- ☞ *Mutual exclusion* — shared resources must be updated atomically
- ☞ *Condition synchronization* — operations may need to be delayed if shared resources are not in an appropriate state (e.g., read from empty buffer)

Liveness — ensuring progress:

- ☞ *No Deadlock* — some process can always access a shared resource
- ☞ *No Starvation* — all processes can eventually access shared resources

Notations for expressing concurrent computation must address:

1. **Process Creation:** how is concurrent execution specified?
2. **Communication:** how do processes communicate?
3. **Synchronization:** how is consistency maintained?

Idioms, Patterns and Architectural Styles

Idioms, patterns and architectural styles express *best practice* in resolving common design problems.

- ❑ Idioms
  - ➪ “a low-level pattern specific to a programming language” *— or more generally: “an implementation technique”*
- ❑ Design patterns
  - ➪ “a commonly-recurring structure of communicating components that solves a general design problem within a particular context”
- ❑ Architectural patterns (styles)
  - ➪ “a fundamental structural organization schema for software systems”

— cf. Buschmann et al., Pattern-Oriented Software Architecture, pp. 12-14

Java

Language design influenced by existing OO languages (C++, Smalltalk ...):

- ❑ Strongly-typed, concurrent, pure object-oriented language
- ❑ Syntax, type model influenced by C++
- ❑ Single-inheritance but multiple subtyping
- ❑ Garbage collection

Innovation in support for network applications:

- ❑ Standard API for language features, basic GUI, IO, concurrency, network
- ❑ Compiled to bytecode; interpreted by portable abstract machine
- ❑ Support for native methods
- ❑ Classes can be dynamically loaded over network
- ❑ Security model protects clients from malicious objects

Java applications do not have to be installed and maintained by users

Threads

A Thread defines its behaviour in its run method, but is started by calling start():

```
// Copyright (c) 1995, 1996 Sun Microsystems, Inc. All Rights Reserved.

class TwoThreadsTest {
   public static void main (String[] args) {
      new SimpleThread("Jamaica").start();    // Instantiate, then start
      new SimpleThread("Fiji").start();
   }
}

class SimpleThread extends Thread {
   public SimpleThread(String str) {
      super(str);                             // Call Thread constructor
   }
   public void run() {                        // What the thread does
      for (int i = 0; i < 10; i++) {
         System.out.println(i + " " + getName());
         try {
            sleep((int)(Math.random() * 1000));
         } catch (InterruptedException e) { }
      }
      System.out.println("DONE! " + getName());
   }
}
```

Running the TwoThreadsTest

```
0 Jamaica
0 Fiji
1 Jamaica
1 Fiji
2 Jamaica
2 Fiji
3 Jamaica
3 Fiji
4 Jamaica
4 Fiji
5 Jamaica
6 Jamaica
5 Fiji
6 Fiji
7 Fiji
7 Jamaica
8 Jamaica
9 Jamaica
8 Fiji
DONE! Jamaica
9 Fiji
DONE! Fiji
```

In this implementation of Java, the execution of the two threads is interleaved.

This is *not* guaranteed for all implementations!

✎ *Why are the output lines never garbled?*

E.g.

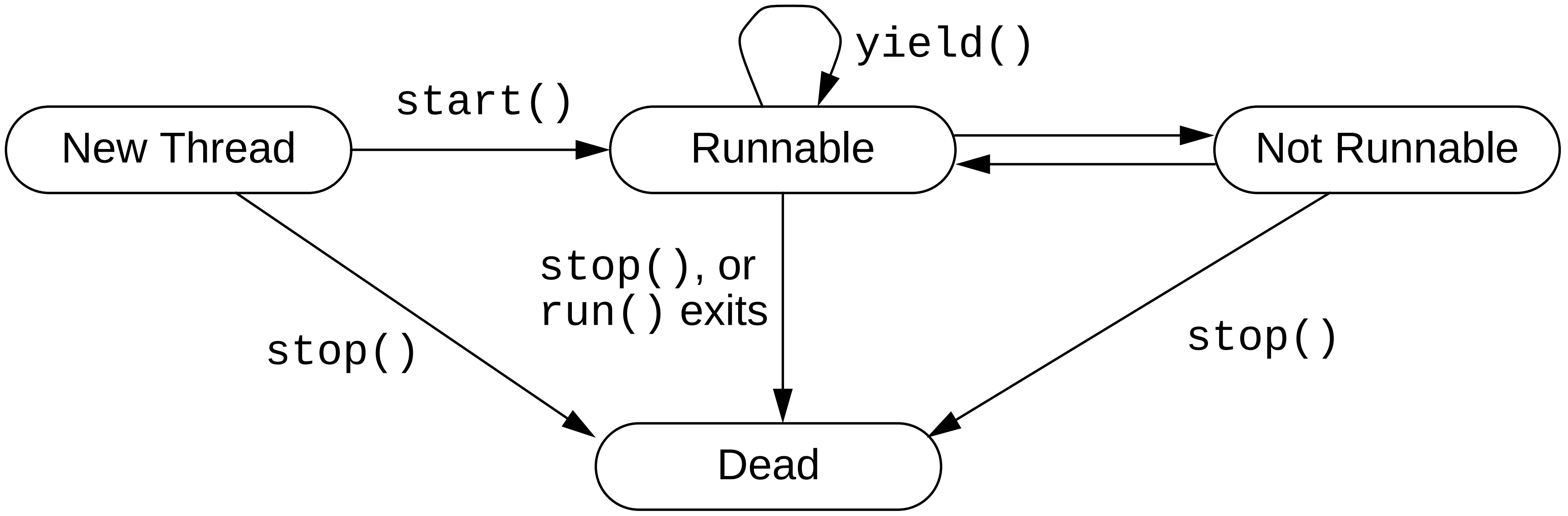
```
00 JaFimajicai
...
```

java.lang.Thread

The Thread class encapsulates all information concerning running threads of control:

```
public class java.lang.Thread
   extends java.lang.Object implements java.lang.Runnable
{
   public Thread();                         // Public constructors
   public Thread(Runnable target);
   public Thread(Runnable target, String name);
   public Thread(String name);
...
   public static void sleep(long millis)// Current thread sleeps
            throws InterruptedException;
   public static void yield();          // Yield control (equal priority)
...
   public final String getName();
   public void run();                   // “main()” method
   public synchronized void start();    // Starts a thread running
   public final void suspend();         // Temporarily halts a thread
   public final void resume();          // Allow to resume after suspend()
   public final void stop();            // Throws a ThreadDeath error
   public final void join()             // Waits for thread to die
            throws InterruptedException;
...
}
```

Transitions between Thread States



Runnable →	*← Not Runnable*
`sleep()`	time elapsed
`suspend()`	`resume()`
`wait()`	`notify()` or `notifyAll()`
blocked on I/O	I/O completed

java.lang.Runnable

Since multiple inheritance is not supported, it is not possible to inherit from both `Thread` and from another class providing useful behaviour (like `Applet`).

In these cases it is sufficient to define a class that implements the Runnable interface, and to call the Thread constructor with an instance of that class as a parameter:

```
public interface java.lang.Runnable
{
   public abstract void run();
}
```

Creating Threads

A `Clock` object updates the time as an `Applet` with its own `Thread`:

```
import java.awt.Graphics; // Copyright (c) 1995, 1996 Sun Microsystems, Inc. All Rights Reserved.
import java.util.Date;

public class Clock extends java.applet.Applet implements Runnable {

   Thread clockThread = null;

   public void start() {
      if (clockThread == null) {
         clockThread = new Thread(this, "Clock");       // NB: creates its own thread
         clockThread.start();
      }
   }

   public void run() {
      // loop terminates when clockThread is set to null in stop()
      while (Thread.currentThread() == clockThread) {
         repaint();
         try { clockThread.sleep(1000); }
         catch (InterruptedException e){ }
      }
   }

   public void paint(Graphics g) {
      Date now = new Date();
      g.drawString(now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds(), 5, 10);
   }

   public void stop() { clockThread = null; }
}
```

Synchronization

Without synchronization, an arbitrary number of threads may be running at any time within the methods of an object.

One can either declare an entire method to be synchronized with other synchronized methods of an object:

```
public class PrintStream extends FilterOutputStream {
   ...
   public synchronized void println(String s);// Only one may run at a time
   public synchronized void println(char c);
   ...
}
```

or an individual block within a method may be synchronized with respect to some object:

```
synchronized (resource) { // Lock resource before using it
   ...
}
```

wait and notify

Sometimes threads must be delayed until a resource is in a suitable state:

```
class Slot {                                 // Implements a one-slot buffer
   private int contents;
   private boolean available = false;        // the condition variable

   public synchronized int get() {           // put contents, if available
      while (available == false) {
         try { wait(); }                     // wait until there is something to get()
         catch (InterruptedException e) { }
      }
      available = false;
      notify();                              // wake up the producer
      return contents;
   }

   public synchronized void put(int value) { // put value, if there is room
      while (available == true) {
         try { wait(); }                     // wait until there is room to put()
         catch (InterruptedException e) { }
      }
      contents = value;
      available = true;
      notify();                              // wake up the consumer
   }
}
```

java.lang.Object

Unlike `synchronized`, `wait()` and `notify()` are methods rather than keywords:

```
public class java.lang.Object
{
   public Object();
   public boolean equals(Object obj);
   public final Class getClass();
   public int hashCode();
   public String toString();
   public final void wait()
            throws InterruptedException, IllegalMonitorStateException;
   public final void wait(long timeout)
            throws InterruptedException, IllegalMonitorStateException;
   public final void wait(long timeout, int nanos)
            throws InterruptedException, IllegalMonitorStateException;
   public final void notify() throws IllegalMonitorStateException;
   public final void notifyAll() throws IllegalMonitorStateException;
   protected Object clone()
            throws CloneNotSupportedException, OutOfMemoryException;
   protected void finalize() throws Throwable;
}
```

Summary

You Should Know The Answers To These Questions:

- ❑ What is the distinction between "concurrency" and "parallelism"?
- ❑ What are classical applications of concurrent programming?
- ❑ Why are concurrent programs more complex than sequential ones?
- ❑ What are "safety" and "liveness"? Give examples.
- ❑ How do you create a new thread in Java?
- ❑ What states can a Java thread be in? How does it change state?
- ❑ When should you declare a method to be `synchronized`?

Can You Answer The Following Questions?

- ✎ *What is an example of a "race condition"?*
- ✎ *When will a concurrent program run faster than an equivalent sequential one? When will it be slower?*
- ✎ *What is the difference between deadlock and starvation?*
- ✎ *What happens if you call* `wait` *or* `notify` *outside a synchronized method or block?*
- ✎ *When is it better to use synchronized blocks rather than methods?*

2. Safety

Overview

- ❑ Immutability:
  - ☞ avoid safety problems by avoiding state changes
- ❑ Full Synchronization:
  - ☞ dynamically ensure exclusive access
- ❑ Partial Synchronization:
  - ☞ restrict synchronization to "critical sections"
- ❑ Containment:
  - ☞ structurally ensure exclusive access

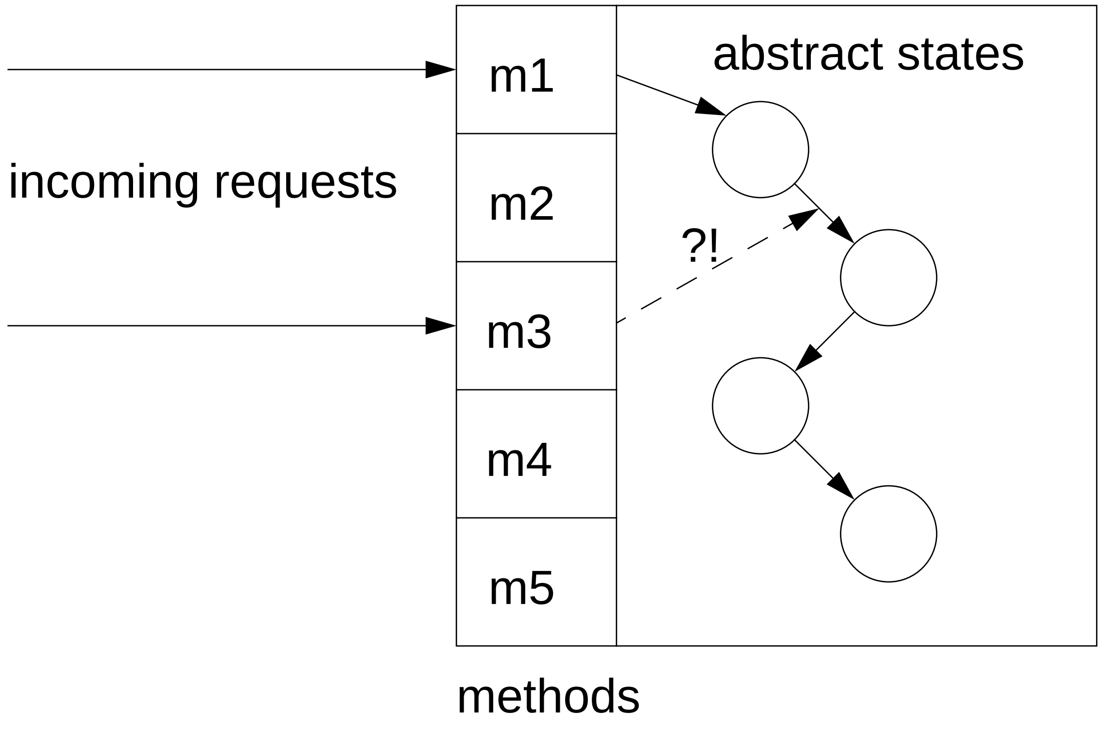
Sources

- ❑ D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996
- ❑ D. Lea, *On-line Supplement to Concurrent Programming in Java,* `http://gee.cs.oswego.edu/dl/cpj/index.html`

Safety problems

Objects must only be accessed when they are in a consistent state

☞ methods must maintain class (state and representation) invariants



Each method may assume that the object is in a “clean”, consistent state when it starts, and it must ensure that it is left in a clean state when it is done.

If methods interleave arbitrarily, an inconsistent state may be accessed, and the object may be left in a “dirty” state.

Immutable classes

Intent

Bypass safety issues by not changing an object’s state after creation.

Applicability

- ❑ When objects represent values of simple ADTs
  - ☞ colours (java.awt.Color), numbers (java.lang.Integer) and strings (java.lang.String)
- ❑ When classes can be separated into mutable and immutable versions
  - ☞ java.lang.String vs. java.lang.StringBuffer
- ❑ When updating by copying is cheap
  - ☞ “hello” + “ ” + “world” $\rightarrow$ “hello world”
- ❑ When multiple instances can represent the same value
  - ☞ i.e., two distinct copies of the integer 712 represent the same value

Immutability variants

Variants

- ❑ Stateless methods
  - ☞ methods that do not access an object’s state do not need to be synchronized (such methods can be declared `static`)
  - ☞ any temporary state should be purely local to the method
- ❑ Stateless objects
  - ☞ an object whose “state” is dynamically computed needs no synchronization
- ❑ “Hardening”
  - ☞ object becomes immutable after a mutable phase
  - ☞ be sure that object is exposed to concurrent threads only after hardening

Immutable classes — design steps

- ❑ Declare a class with instance variables that are never changed after construction.

```
class Relay {                           // a within-package helper for some Server class
   private final Server server_;// “blank final” (in Java 1.1)
   Relay(Server s) {                    // blank finals must be initialized
      server_ = s;                      // in all constructors
   }
   void doIt() {
      server_.doIt();
   }
}
```

- ❑ Especially if the class represents an immutable data abstraction (such as `String`), consider overriding `Object.equals` and `Object.hashCode`.
- ❑ Consider writing methods that generate new objects of this class. (e.g., `String` concatenation)
- ❑ Consider declaring the class as `final`.
- ❑ If only some variables are immutable, use synchronization or other techniques for the methods that are not stateless.

Fully Synchronized Objects

Intent

Maintain consistency by fully synchronizing all methods.At most one method will run at any point in time.

Applicability

- ❑ You want to *eliminate all possible* read/write and write/write *conflicts*, regardless of the context in which it the object is used.
- ❑ All methods can *run to completion* without waits, retries, or infinite loops.
- ❑ You *do not need* to use instances in *a layered design* in which other objects control synchronization of this class.
- ❑ You can avoid or deal with liveness failures, for example, by:
  - ☞ Exploiting partial immutability
  - ☞ Removing synchronization for accessors.
  - ☞ Removing synchronization in invocations.
  - ☞ Arranging per-method concurrency.

Fully Synchronization — design steps

- ❑ Declare all methods as `synchronized`
    - ☞ Do not allow any direct access to state (i.e, no `public` instance variables; no methods that return references to instance variables).
    - ☞ Constructors cannot be marked as `synchronized` in Java. Use a synchronized block in case a constructor passes `this` to multiple threads.
    - ☞ Methods that access `static` variables must either do so via `static synchronized` methods or within blocks of the form `synchronized(getClass()) { ... }`.
- ❑ Ensure that every `public` method exits leaving the object in a consistent state, even if it exits via an exception.
- ❑ Keep methods short so they can atomically run to completion. State-dependent actions must rely on *balking:*
    - ☞ Return failure (i.e., exception) to client if preconditions fail
    - ☞ If the precondition does not depend on state (e.g., just on the arguments), then no need to run check in synchronized code!
    - ☞ Provide public accessor methods so that clients can check conditions before making request!

Example: a BalkingBoundedCounter

A Bounded Counter holds a value between MIN and MAX.

If the preconditions for `inc()` or `dec()` fail, an exception is raised:

```
public class BalkingBoundedCounter {
   protected long count_ = BoundedCounter.MIN;

   public synchronized long value() { return count_; }

   public synchronized void inc() throws CannotIncrementException {
      if (count_ >= BoundedCounter.MAX)
         throw new CannotIncrementException();
      else
         ++count_;
   }

   public synchronized void dec() throws CannotDecrementException {
      if (count_ <= BoundedCounter.MIN)
         throw new CannotDecrementException();
      else
         --count_;
   }
}
```

✎ *What safety problems would arise if this class were not fully synchronized?*

Example: an ExpandableArray

This Expandable Array is a simplified variant of java.util.Vector:

```
import java.util.NoSuchElementException;

public class ExpandableArray {
   private Object[] data_;                         // the elements
   private int size_;                              // the number of slots used

   public ExpandableArray(int cap) {
      data_ = new Object[cap];                     // reserve some space
      size_ = 0;
   }

   public synchronized int size() { return size_; }

   public synchronized Object at(int i)            // subscripted access
      throws NoSuchElementException {
      if (i < 0 || i >= size_ )
         throw new NoSuchElementException();
      else
         return data_[i];
   }

   ...
```

```
public class ExpandableArray {
   ...
   public synchronized void append(Object x) {   // add at end
      if (size_ >= data_.length) {               // need a bigger array
         Object[] olddata = data_;
         data_ = new Object[3 * (size_ + 1) / 2];// increase by ~ 50%
         for (int i = 0; i < size_; ++i)
            data_[i] = olddata[i];
      }
      data_[size_++] = x;
   }
   public synchronized void removeLast()
      throws NoSuchElementException {
      if (size_ == 0)
         throw new NoSuchElementException();
      else
         data_[--size_] = null;
   }
}
```

✎ *What could happen if any of these methods were not synchronized?*

Bundling Atomicity

- Consider adding synchronized methods that perform frequently desired *sequences* of actions as single atomic action, so that clients do not need to impose extra synchronization or control.

```
public interface Procedure {          // apply some operation to a single object
   public void apply(Object x);
}
public class ExpandableArrayV2 extends ExpandableArray {
   public ExpandableArrayV2(int cap) { super(cap); }
   public synchronized void applyToAll(Procedure p) {
      for (int i = 0; i < size_; ++i) { // oops -- SIZE _ and data_
         p.apply(data_[i]);             // should have been protected!
      }
   }
```

✎ *What possible liveness problems does this technique introduce?*

Inner classes

Anonymous inner classes (in Java 1.1) are the OO equivalent of lambda expressions:

```
class ExpandbleArrayUser {
   public static void main(String[] args) {
      ExpandableArrayV2 a = new ExpandableArrayV2(100);
      for (int i = 0; i < 100; ++i)     // fill it up
         a.append(new Integer(i));
      a.applyToAll(new Procedure {      // print all elements
            public void apply(Object x) { System.out.println(x); }
         }
      )
   }
}
```

Any variables shared with the host object must be declared `final` *(immutable).*

Partial Synchronization

Intent

Reduce overhead by synchronizing only within "critical sections".

Applicability

- ❑ When objects have both mutable and immutable instance variables.
- ❑ When methods can be split into a "critical section" that deals with mutable state and a part that does not.

Design steps

- ❑ Fully synchronize all methods
- ❑ Remove synchronization for *accessors to atomic or immutable* values
- ❑ Remove synchronization for methods that access mutable state through a single other, *already synchronized* method
- ❑ Replace method synchronization by block synchronization for methods where access to mutable state is restricted to a *single, critical section*

Example: LinkedCells

```
public class LinkedCell {
   protected double value_;                            // doubles are not atomic!
   protected final LinkedCell next_;                   // fixed

   public LinkedCell (double v, LinkedCell t) { value_ = v; next_ = t; }

   public synchronized double value() { return value_; }

   public synchronized void setValue(double v) { value_ = v; }

   public LinkedCell next() { return next_; }      // no synch needed

   public double sum() {                               // add up all element values
      double v = value();                              // get value via
      if (next() != null)                              // synchronized accessor
         v += next().sum();
      return v;
   }

   public boolean includes(double x) {                 // search for x
      synchronized(this) {                             // synch to access value
         if (value_ == x) return true;
      }
      if (next() == null) return false;
      else return next().includes(x);
   }
}
```

Containment

Intent

Achieve safety by avoiding shared variables. Unsynchronized objects are "contained" inside other objects that have at most one thread active at a time.

Applicability

- ❑ There is no need for shared access to the embedded objects.
- ❑ The embedded objects can be conceptualized as exclusively held resources
- ❑ You can tolerate the additional context dependence for embedded objects.
- ❑ Embedded objects must be structured as *islands* — communication-closed sets of objects ultimately reachable from a single unique reference. They cannot contain methods that reveal their identities to other objects.
- ❑ You are willing to hand-check designs for compliance.
- ❑ You can deal with or avoid indefinite postponements or deadlocks in cases where host objects must transiently acquire multiple resources.

Contained Objects — design steps

- ❑ Define the interface for the outer host object.
  - ☞ The host could be, e.g., an Adaptor, a Composite, or a Proxy, that provides synchronized access to an existing, unsynchronized class
- ❑ Ensure that the host is either fully synchronized, or is in turn a contained object.
- ❑ Define instances variables that are *unique* references to the contained objects.
  - ☞ Make sure that these references cannot leak outside the host!
  - ☞ Establish policies and implementations that ensure that acquired references are really unique!
  - ☞ Consider methods to duplicate or clone contained object, to ensure that copies are unique

Managed Ownership

- ❑ Model contained objects as physical resources:
  - ☞ If you have one, then you can do something that you couldn't do otherwise.
  - ☞ If you have one, then no one else has it.
  - ☞ If you give one to someone else, then you no longer have it.
  - ☞ If you destroy one, then no one will ever have it.
- ❑ If contained objects can be passed among hosts, define a transfer protocol.
  - ☞ Hosts should be able to *acquire*, *give*, *take*, *exchange* and *forget* resources
  - ☞ Consider using a dedicated class to manage transfer

A minimal transfer protocol class

This class is essentially a one-slot buffer for transferring resources between hosts in separate threads.

```
public class ResourceVariable {
   protected Object ref_;
   public ResourceVariable(Object res) { ref_ = res; }
   public synchronized Object resource() { return ref_; }
   public synchronized Object exchange(Object r) {
      Object old = ref_;
      ref_ = r;
      return old;
   }
}
```

NB: `exchange()` is enough to implement most transfer operations, e.g., `take()` is implemented by `exchange(null)`

Summary

You Should Know The Answers To These Questions:

- ❑ Why are immutable classes inherently safe?
- ❑ Why doesn’t a “relay” need to be synchronized?
- ❑ What is “balking”? When should a method balk?
- ❑ When is partial synchronization better than full synchronization?
- ❑ How does containment avoid the need for synchronization?

Can You Answer The Following Questions?

- ✎ *When is it all right to declare only some methods as* `synchronized`*?*
- ✎ *When is an inner class better than an explicitly named class?*
- ✎ *What liveness problems can full synchronization introduce?*
- ✎ *Why is it a bad idea to have two separate critical sections in a single method?*
- ✎ *Does it matter if a contained object is synchronized or not?*

3. State-dependent Action

Overview

- ❑ Liveness and Fairness
  - ☞ The Dining Philosophers problem
- ❑ Guarded Methods
  - ☞ Checking guard conditions
  - ☞ Handling interrupts
  - ☞ Structuring notification
  - ☞ Tracking state
  - ☞ Delegating notifications

Sources

- ❑ A. Burns and G. Davies, *Concurrent Programming*, Addison-Wesley, 1993
- ❑ D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996
- ❑ D. Lea, *On-line Supplement to Concurrent Programming in Java,* `http://gee.cs.oswego.edu/dl/cpj/index.html`

Liveness Problems

Liveness properties guarantee that your (concurrent) programs will make progress.

A program may be “safe”, yet suffer from various kinds of liveness problems:

❑ Contention:

☞ AKA “starvation” or “indefinite postponement” — the system as a whole makes progress, but some individual processes don’t

❑ Dormancy:

☞ A waiting process fails to be woken up

❑ Deadlock:

☞ Two or more processes are blocked, waiting for resources held by the others (i.e., in a cycle)

❑ Premature termination:

☞ A process is killed before it should be

Achieving Liveness

There are various strategies and techniques to ensure liveness:

- ❑ Start with safe design and selectively remove synchronization
- ❑ Start with live design and selectively add safety
- ❑ Adopt design patterns that limit the need for synchronization
- ❑ Adopt standard architectures that avoid cyclic dependencies

The Dining Philosophers Problem

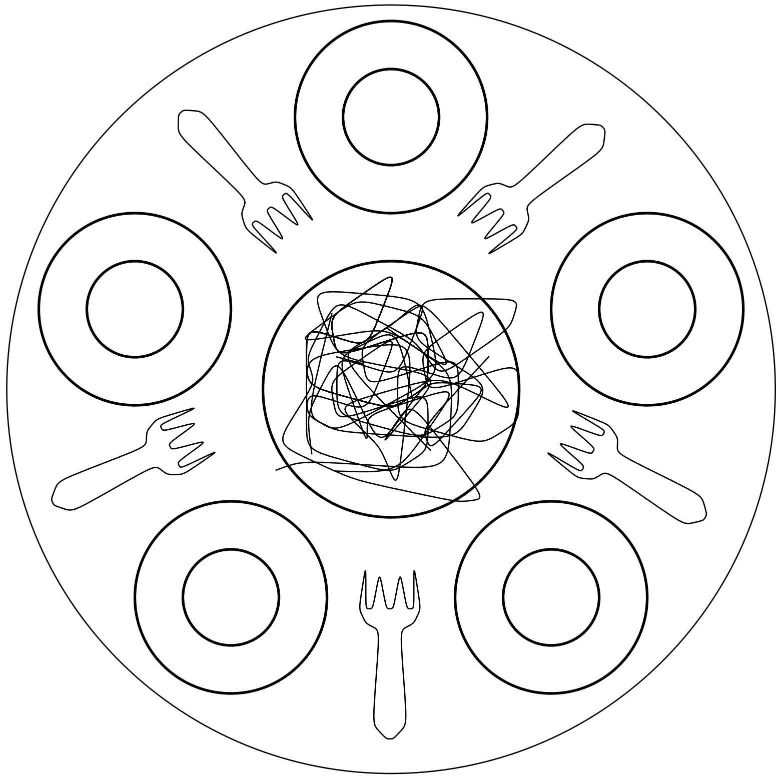
Philosophers alternate between thinking and eating.

A philosopher needs two forks to eat.

No two philosophers may hold the same fork simultaneously.

There should be no deadlock and no starvation.

Want efficient behaviour under absence of contention.



Dining Philosophers, Safety and Liveness

Dining Philosophers illustrates many classical safety and liveness issues:

- ❑ *Mutual Exclusion:* Each chopstick can be used by one philosopher at a time
- ❑ *Condition synchronization:* A philosopher needs two forks to eat
- ❑ *Shared variable communication:* Philosophers share forks ...
- ❑ *Message-based communication:* ... or they can pass forks to each other
- ❑ *Busy-waiting:* A philosopher can poll for forks ...
- ❑ *Blocked waiting:* ... or can sleep till woken by a neighbour
- ❑ *Livelock:* All philosophers can grab the left fork and busy-wait for the right ...
- ❑ *Deadlock:* ... or grab the left one and wait (sleep) for the right
- ❑ *Starvation:* A philosopher may starve if the left and right neighbours are always faster at grabbing the forks

Dining Philosopher Solutions

There are countless solutions to the Dining Philosophers problem that use various concurrent programming styles and patterns, and offer varying degrees of liveness guarantees:

- ❑ Number the forks;
  philosophers grab the lowest numbered fork first.
- ❑ Have philosophers leave the table while they think;
  allow at most four to sit at a time;
  philosophers queue to sit down.

✎ *Is deadlock possible in either case?*
✎ *What about starvation?*
✎ *Are these solutions “fair”?*

Fairness

There are subtle differences between definitions of fairness:

Weak fairness:

- ☞ If a process *continuously* makes a request, *eventually* it will be granted.

Strong fairness:

- ☞ If a process makes a request *infinitely often*, *eventually* it will be granted.

Linear waiting:

- ☞ If a process makes a request, it will be granted before any other process is granted the request more than once.

FIFO (first-in first out):

- ☞ If a process makes a request, it will be granted before that of any process making a later request.

Guarded Methods

Intent

Temporarily suspend an incoming thread when an object is not in the right state to fulfil a request, and wait for the state to change rather than balking (raising an exception).

Applicability

- ❑ Clients can tolerate indefinite postponement. (Otherwise, use a *balking design*.)
- ❑ You can guarantee that the required states are eventually reached (via other requests), or if not, that it is acceptable to block forever.
- ❑ You can arrange that notifications occur after all relevant state changes. (Otherwise consider a design based on a *busy-wait spin loop*.)
- ❑ You can avoid or cope with liveness problems due to waiting threads retaining all synchronization locks (except for that of the host).
- ❑ You can construct computable predicates describing the state in which actions will succeed. (Otherwise consider an *optimistic design*.)
- ❑ Conditions and actions are managed within a single object. (Otherwise consider a *transactional form*.)
- ❑ You have no need to encapsulate waiting and notification mechanics within special *condition objects*.

Guarded Methods — design steps

The basic recipe is to use `wait` in a conditional loop to block until it is safe to proceed, and use `notifyAll` to wake up blocked threads.

```
public synchronized Object service() {
   while (wrong State) {
      try { wait(); }
      catch (InterruptedException e) { }
   }
   // fill request
   // and change state
   notifyAll();                  // NB: use notify() only if it does not matter
   return result;                // which waiting thread you wake up
}
```

Separate interface from policy

❑ Define interfaces for the methods, so that classes can implement guarded methods according to different policies.

```
public interface BoundedCounter {
   public static final long MIN = 0;  // minimum allowed value
   public static final long MAX = 10; // maximum allowed value

   public long value();               // invariant: MIN <= value() <= MAX
                                      // initial condition: value() == MIN

   public void inc();                 // increment only when value() < MAX
   public void dec();                 // decrement only when value() > MIN
}
```

Check guard conditions

- ❑ Define a predicate that precisely describes the conditions under which actions may proceed. (This can be encapsulated as a helper method.)

- ❑ Precede the conditional actions with a guarded wait loop of the form:

  ```
  while (!condition)
  try { wait(); } catch (InterruptedException ex) { ... }
  ```

  Optionally, encapsulate this code as a helper method.

- ❑ If there is only *one possible condition* to check in this class (and all plausible subclasses), and notifications are issued only when the condition is true, then there is no need to re-check the condition after returning from `wait()`

- ❑ Ensure that the object is in a consistent state (i.e., the class invariant holds) before entering any `wait` (since wait releases the synchronization lock). The easiest way to do this is to perform the guards *before* taking any actions.

Handle interrupts

- ❑ Establish a policy for dealing with `InterruptedExceptions` (which will also force a return from `wait`). Possible policies are:

  - ☞ Ignore interrupts (i.e., have an empty `catch` clause), which preserves safety at the possible expense of liveness.
  - ☞ Terminate the current thread (via `stop`). This also preserves safety, though brutally!
  - ☞ Exit the method, possibly raising an exception. This preserves liveness but may require the caller to take special action to preserve safety.
  - ☞ Take some pre-planned action; such as cleanup and restart.
  - ☞ Ask for user intervention before taking further action.

Interrupts can be useful to signal that the guard can never become true because, for example, the collaborating threads have terminated.

Signal state changes

- ❑ *Add notification code* to each method of the class that changes state in any way that can affect the value of a guard condition. Some options are:
  - ☞ `notifyAll` wakes up all threads that are blocked in waits for the host object. Calls to `notifyAll` (as well as `notify`) *must* be enclosed within a synchronized method or block.
  - ☞ `notify` wakes up only one thread (if any exist). This is best treated as an optimization where (i) all blocked threads are necessarily waiting for conditions signalled by the same notifications, (ii) only one of them can be enabled by any given notification, and (iii) it does not matter which one of them becomes enabled.
  - ☞ You build your own special-purpose notification methods using `notify` and `notifyAll`. (For example, to selectively notify threads, or to provide certain *fairness* guarantees.)

Structure notifications

- Ensure that each wait is balanced by at least one notification. Options include:
  - ☞ *Blanket Notifications:* Place a notification at the end of every method that can cause any state change (i.e., assigns any instance variable). While simple and reliable, this approach can cause performance problems ...
  - ☞ *Encapsulating Assignment:* Encapsulate assignment to each variable mentioned in any guard condition in a helper method that performs the notification after updating the variable.
  - ☞ *Tracking state:* Only issue notifications for the particular state changes that could actually unblock waiting threads. This approach may improve performance, at the cost of flexibility. (I.e., subclassing becomes harder.)
  - ☞ *Tracking State Variables:* Maintain an instance variable that represents control state. After each method that changes state, invoke a helper method that re-evaluates the variable and checks if the state change might affect a guard condition, and if so, issues a notification.
  - ☞ *Delegating Notifications:* If aspects of state are maintained by completely contained helper objects, have these helpers issue the notifications.

Encapsulating assignment

```
public class BoundedCounterV0 implements BoundedCounter {
   protected long count_ = MIN;
   public synchronized long value() { return count_; }
   public synchronized void inc() {
      awaitIncrementable();
      setCount(count_ + 1);
   }
   public synchronized void dec() {
      awaitDecrementable();
      setCount(count_ - 1);
   }
   protected synchronized void setCount(long newValue) {
      count_ = newValue;
      notifyAll();              // wake up any thread depending on new value
   }
   protected synchronized void awaitIncrementable() {
      while (count_ >= MAX) try { wait(); } catch(InterruptedException ex) {};
   }
   protected synchronized void awaitDecrementable() {
      while (count_ <= MIN) try { wait(); } catch(InterruptedException ex) {};
   }
}
```

Tracking State

The only transitions that could possibly affect waiting threads in BoundedCounter are those that step away from logical states *bottom* and *top*:

```
public class BoundedCounterVST implements BoundedCounter {
   protected long count_ = MIN;
   public synchronized long value() {
      return count_;
   }
   public synchronized void inc() {
      while (count_ == MAX)
         try { wait(); } catch(InterruptedException ex) {};
      if (count_++ == MIN)                   // signal if previously in bottom state
         notifyAll();
   }
   public synchronized void dec() {
      while (count_ == MIN)
         try { wait(); } catch(InterruptedException ex) {};
      if (count_-- == MAX)                   // signal if previously in top state
         notifyAll();
   }
}
```

Tracking State Variables

```
public class BoundedCounterVSW implements BoundedCounter {
   static final int BOTTOM= 0;                     // logical states
   static final int MIDDLE= 1;
   static final int TOP    = 2;

   protected int state_ = BOTTOM;                  // the state variable
   protected long count_ = MIN;

   protected synchronized void checkState() {
      int oldState = state_;
      if (count_ == MIN)          state_ = BOTTOM;
      else if (count_ == MAX)     state_ = TOP;
      else                        state_ = MIDDLE;

      if (state_ != oldState &&                    // notify on transition
            (oldState == TOP || oldState == BOTTOM))
         notifyAll();
   }

   public synchronized long value() { return count_; }

   public synchronized void inc() {
      while (state_ == TOP)                        // only consult logical state
         try { wait(); } catch(InterruptedException ex) {};
      ++count_;
      checkState();
   }                                               // dec() is similar ...
```

Delegating notifications

`NotifyingLong` class can be used to issue a `notifyAll` to any observer object whenever it changes value:

```
public class NotifyingLong {
   private long value_;
   private Object observer_;
   public NotifyingLong(Object o, long v) {
      observer_ = o;
      value_ = v;
   }
   public synchronized long value() { return value_; }
   public void setValue(long v) {
      synchronized(this) {
         value_ = v;
      }
      synchronized(observer_) {
         observer_.notifyAll();
      }
   }
}
```

Delegating notifications ...

The resulting `BoundedCounter` class differs from an Adapter only in that the helper object provides the change notifications on behalf of the host:

```
public class BoundedCounterVNL implements BoundedCounter {
   private NotifyingLong c_ = new NotifyingLong(this, MIN);
   public synchronized long value() {
      return c_.value();
   }
   public synchronized void inc() {
      while (c_.value() >= MAX)
         try { wait(); } catch(InterruptedException ex) {};
      c_.setValue(c_.value()+1);
   }
   public synchronized void dec() {
      while (c_.value() <= MIN)
         try { wait(); } catch(InterruptedException ex) {};
      c_.setValue(c_.value()-1);
   }
}
```

Using template methods

To improve extensibility, consider separating out the code that (unconditionally) performs the action in a separate (non-public) method. The same action code can then be used within different guarded methods:

```
public class BoundedCounterVSG implements BoundedCounter {
   protected long count_ = MIN;
   // non-public actions
   protected long value_() { return count_; }
   protected void inc_() { ++count_; }
   protected void dec_() { --count_; }
   // possibly guarded public methods
   public synchronized long value() {
      return value_();
   }
   public synchronized void inc() {
      while (value_() >= MAX)
         try { wait(); } catch(InterruptedException ex) {};
      inc_();
      notifyAll();
   }
   // etc ....
```

Summary

You Should Know The Answers To These Questions:

- ❑ What kinds of liveness problems can occur in concurrent programs?
- ❑ What is the difference between livelock and deadlock?
- ❑ When should methods recheck guard conditions after waking from a `wait()`?
- ❑ Why should you usually prefer `notifyAll()` to `notify()`?
- ❑ When and where should you issue notification?

Can You Answer The Following Questions?

- ✎ *How can you detect deadlock? How can you avoid it?*
- ✎ *What is the easiest way to guarantee fairness?*
- ✎ *When are guarded methods better than balking?*
- ✎ *What is the best way to structure guarded methods for a class if you would like it to be easy for others to define correctly functioning subclasses?*
- ✎ *Is the complexity of delegating notifications worth it?*

4. Asynchronous Methods

Overview

- ❑ Asynchronous invocations
- ❑ Simple Relays
  - ☞ Direct Invocations
  - ☞ Thread-based messages; Gateways
  - ☞ Command-based messages
- ❑ Tail calls
- ❑ Early replies
- ❑ Futures

Sources

- ❑ D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996
- ❑ Doug Lea, *On-line Supplement to Concurrent Programming in Java,* `http://gee.cs.oswego.edu/dl/cpj/index.html`

Asynchronous Invocations

Intent

Avoid waiting for a request to be serviced by decoupling sending from receiving.

Applicability

- ❑ When a host object can distribute services amongst multiple helper objects.
- ❑ When an object does not need the result of an invocation to continue doing useful work, for example:
  - ☞ Notifications: inform the target object of a certain state of affairs
  - ☞ Activations: construct and start daemon objects may never terminate
  - ☞ Multicast: invoke independent services on a group of helpers
  - ☞ Relays: hand off work to be performed by a helper
- ❑ When invocations that are *logically* asynchronous, regardless of whether they are coded using threads.
- ❑ During refactoring, when classes and methods are split in order to increase concurrency and reduce liveness problems.

Asynchronous Invocations — form

Generally, asynchronous invocation designs take the following form:

```
class Host {
   public service() {
      pre();               // code that must execute before invocation
      invokeHelper();      // the invocation
      during();            // code that may run at the same time as invocation
      post();              // code that must execute after completion
   }
}
```

Asynchronous Invocations — design steps

Consider the following issues:

- ❑ Does the Host need to get results back from the Helper?
  - ☞ Not if, e.g., the Helper returns results directly to the Host's caller!
- ❑ Can the Host process new requests while the Helper is running?
  - ☞ Might depend on the kind of request ...
- ❑ Does the Host *need* to do something while the Helper is running?
  - ☞ I.e., in the `during` code
- ❑ Does the Host need to do synchronized pre-invocation processing?
  - ☞ I.e., if `service()` is guarded or if `pre()` updates the Host's state
- ❑ Does the Host need to do synchronized post-invocation processing?
  - ☞ I.e., if `post()` updates the Host's state
- ❑ Does post-invocation processing only depend on the Helper's result?
  - ☞ ... or does the host have to wait for other conditions?
- ❑ Is the same Helper always used?
  - ☞ Is a new one generated to help with each new service request?

Simple Relays

A *relay method* is obtains all its functionality by delegating to the helper, without any `pre()`, `during()`, or `post()` actions.

Three common forms:

- ❑ Direct invocations
  - ☞ Invoke the Helper directly, but without synchronization
- ❑ Thread-based massages
  - ☞ Create a new thread to invoke the Helper
- ❑ Command-based messages
  - ☞ Pass the request as a Command object to another object that will run it

Relays are commonly seen in Adaptors.

Direct invocations

Although the invocation is not strictly asynchronous, we have many of the same benefits since the Host is free to accept other requests, while the Host's caller must wait for the reply. This is a special case of *stateless method*.

```
class Host {
   protected Helper helper_ = new Helper();
   public void service() {                                   // no synch
      invokeHelper();
   }
   protected void invokeHelper() { helper_.help(); }         // no synch
}
```

If `helper_` is mutable, it may be necessary to protect it with an accessor:

```
   protected synchronized Helper helper() { return helper_; }
   public void service() {                                   // no synch
      helper().help();
   }
```

Thread-based massages

The invocation can be performed within a new thread:

```
protected void invokeHelper() {
   Runnable r = new Runnable {                  // NB: an inner class
      final Helper h_ = helper_;
      public void run() { h_.help() ; }
   }
   new Thread(r).start();
}
```

The cost of evaluating Helper.help() should outweigh the overhead of creating a thread!

- ☞ If the Helper is a daemon (loops endlessly)
- ☞ If the Helper does IO
- ☞ Possibly, if multiple helper methods are invoked

NB: If `helper_` is mutable, `service()` should be synchronized

Thread-per-message Gateways

Variant: the host may construct a new Helper to service each request.

```
public class FileIO {
   public void writeBytes(String fileName, byte[] data) {
      new Thread (new FileWriter(fileName, data)).start();
   }
   public void readBytes(String fileName, byte[] data) {
      new Thread (new FileReader(fileName, data)).start();
   }
}
class FileWriter implements Runnable {
   private String nm_;                                 // hold arguments
   private byte[] d_;
   public FileWriter(String name, byte[] data) {
      nm_ = name;
      d_ = data;
   }
   public void run() {
      // write bytes in d_ to file nm_ ...
   }
}                                                      // etc.
```

Command-based messages

Instead of invoking the Helper directly, or starting a thread to do so, the Host can put a message in a queue to be read by another object that will invoke the Helper:

```
protected EventQueue q_;
protected invokeHelper() {
   q_.put(new HelperMessage(helper_));
}
```

Command-based forms are most appropriate when you need to support special scheduling, undo, or replay capabilities, or are transporting messages over networks.

Tail calls

Tail-call designs apply when the helper method can be performed as the *last* statement(s) of a method; i.e., when there is no `post()` processing. All `pre()` code can be performed under synchronization, released before the call. The host is then immediately available to accept other messages after issuing the helper invocation.

Observer designs often take this form, where Host is `Subject`, service is `updateState`, pre is `doUpdate`, Helper is `Observer`, and help is `changeNotification`:

```
class Subject {
   protected Observer obs_ = new ...;
   protected double state_;
   public void updateState(double d) {          // not synched
      doUpdate(d);                              // synched
      sendNotification();                       // not synched
   }
   protected synchronized doUpdate(double d) { state_ = d; }
   protected void sendNotification() { obs_.changeNotification(this); }
}
```

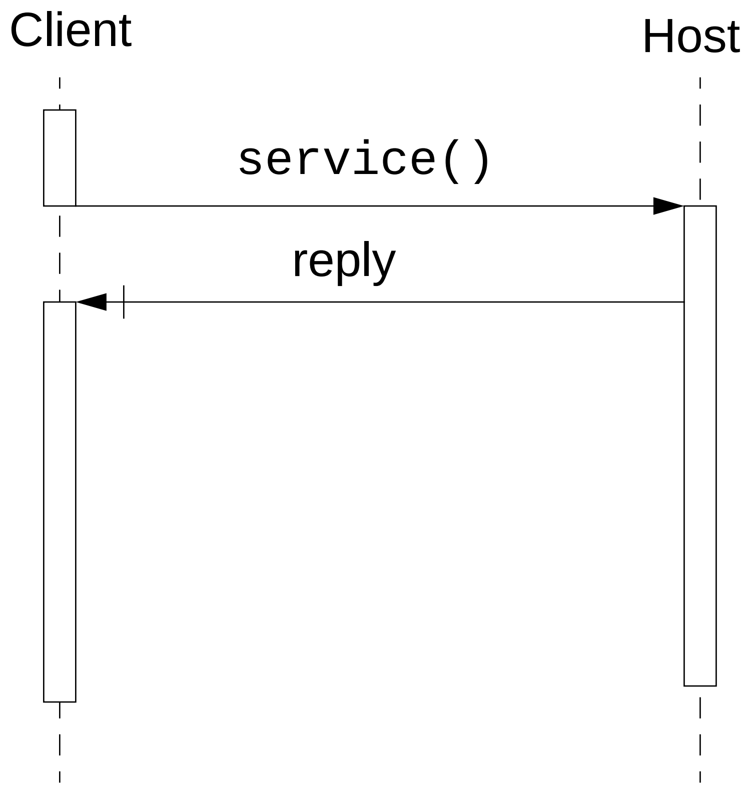
Tail calls with new threads

Rather than direct invocations, the tail call may be performed in a thread. For example:

```
public synchronized void updateState(double d) {
   state_ = d;
   Runnable r = new Runnable {
      final Observer o_ = obs_;
      public void run() { o_.changeNotification(Subject.this); }
   }
   new Thread(r).start();
}
```

Early Reply

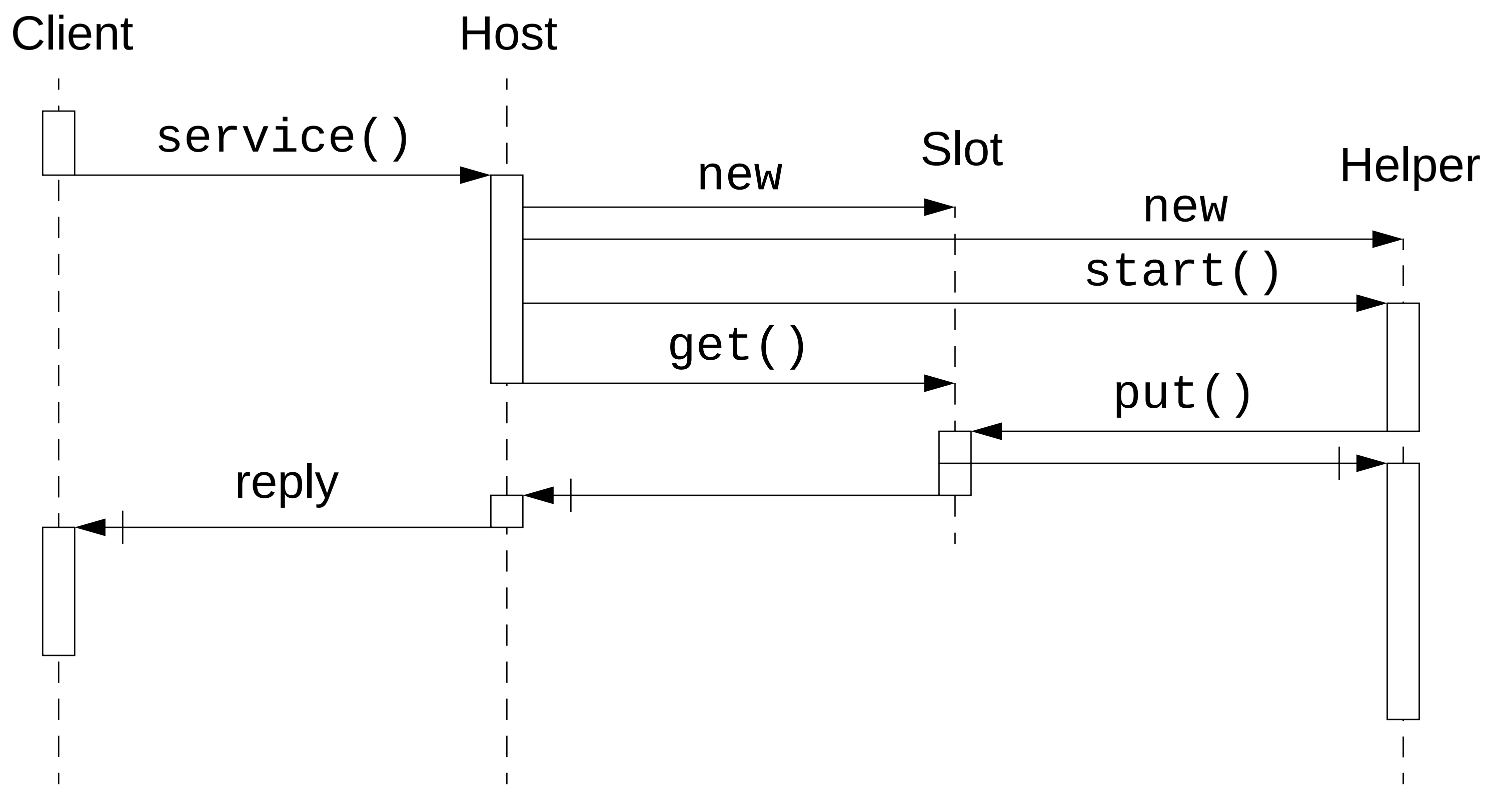
Early reply allows a host to perform useful activities after returning a result to the client:



Early reply is a built-in feature in some programming languages.
It can be easily simulated when it is not a built-in feature.

Simulating Early Reply

A one-slot buffer can be used to pick up the reply from a helper thread:



A one-slot buffer is a simple abstraction that can be used to implement many higher-level concurrency abstractions ...

A One-Slot Buffer

```
class Slot {                                    // a one-slot buffer
   private Object slotVal_;                     // initially null
   public synchronized void put(Object val) {
      while (slotVal_ != null) {
         try { wait(); }
         catch (InterruptedException e) { }
      }
      slotVal_ = val;
      notifyAll();                              // same as notify(),
      return;                                   // if only one producer
   }                                            // and one consumer
   public synchronized Object get() {
      Object rval;
      while (slotVal_ == null) {
         try { wait(); }
         catch (InterruptedException e) { }
      }
      rval = slotVal_;
      slotVal_ = null;
      notifyAll();
      return rval;
   }
}
```

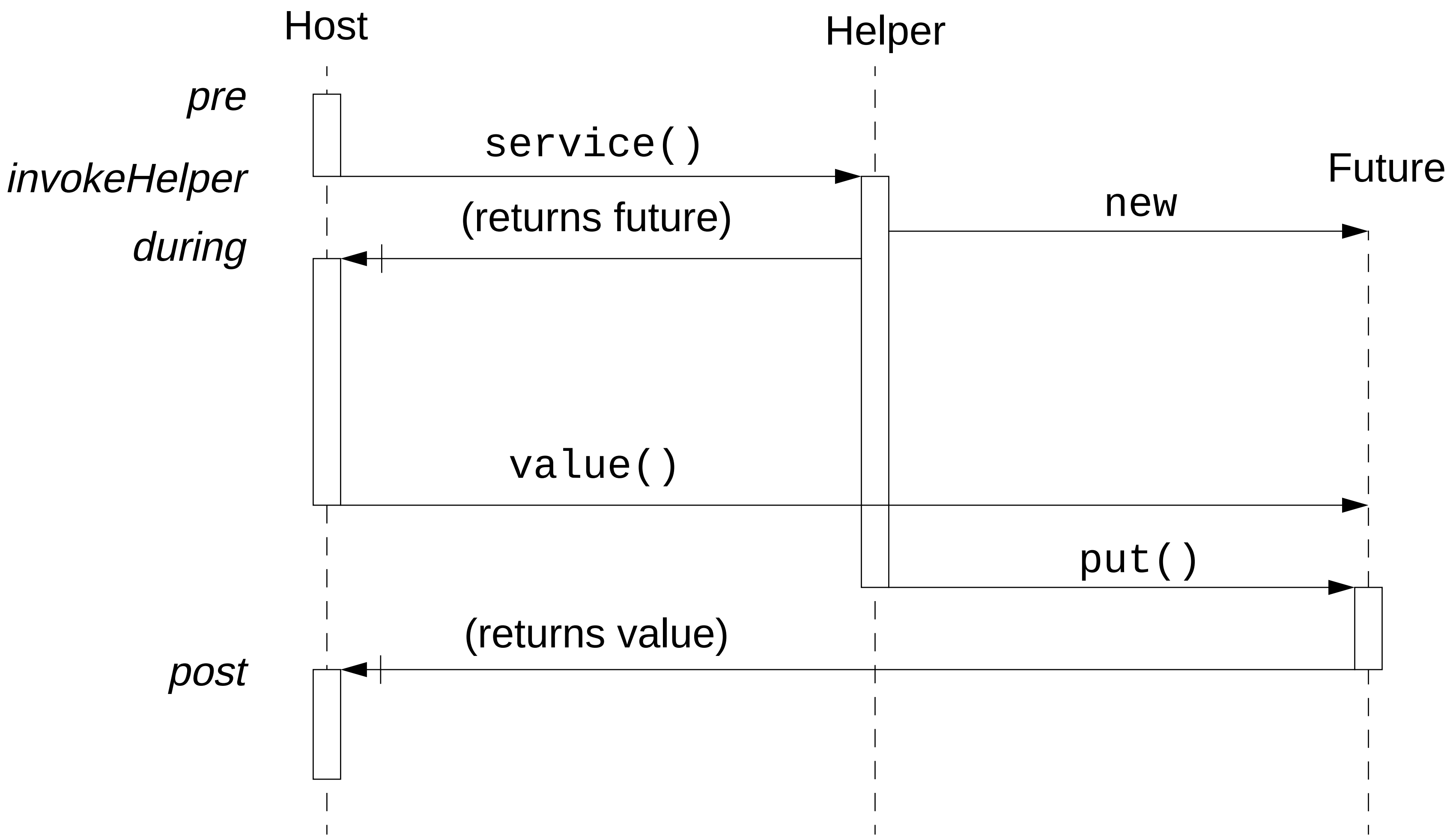
Early Reply in Java

The Helper thread can be easily implemented using an anonymous “inner” class:

```
public static Stuff service() {
   final Slot reply = new Slot();     // NB: shared variable must be immutable
   new Thread(new Runnable () {       // anonymous inner class
      public void run() {
         Stuff result;
         // compute result
         reply.put(result);           // early reply via shared Slot
         // do other stuff
      }
   }).start();
   return (Stuff) reply.get();        // wait till reply is ready
}
```

Futures

Futures allow a host to continue in parallel with a helper until the future value is needed:



A Future Class

Futures can be implemented as a layer of abstraction around a shared Slot:

```
class Future {
   private Object val_;          // initially null
   private Slot slot_;           // shared with some worker
   public Future(Slot slot) {
      slot_ = slot;
   }
   public Object value() {
      if (val_ == null)
         val_ = slot_.get();     // be sure to only get() once!
      return val_;
   }
}
```

Using Futures in Java

WIth futures, the client, rather than the host, proceeds in parallel with a helper thread.

```
public static Future service () {
   final Slot slot = new Slot();       // immutable shared slot
   new Thread(new Runnable () {        // start anonymous inner helper
      public void run() {
         // compute result
         slot.put(result);             // pass result to Future
      }
   }).start();
   return new Future(slot);            // immediately return Future
}                                      // client will wait when result needed
```

Without special language support, futures are less transparent than early replies, since the client must explicitly request a `value()` from the future object.

Summary

You Should Know The Answers To These Questions:

- ❑ What general form does an asynchronous invocation take?
- ❑ When should you consider using asynchronous invocations?
- ❑ In what sense can a direct invocation be “asynchronous”?
- ❑ Why (and how) would you use “inner classes” to implement asynchrony?
- ❑ What is “early reply”, and when would you use it?
- ❑ What are “futures”, and when would you use them?
- ❑ How can implement futures and early replies in Java?

Can You Answer The Following Questions?

- ✎ *Why are servers commonly structured as thread-per-message gateways?*
- ✎ *Which of the concurrency abstractions we have discussed till now can be implemented using one-slot-buffers as the only synchronized objects?*
- ✎ *When are futures better than early replies? Vice versa?*

5. Fine-grained Synchronization

Overview

- ❑ Condition Objects
- ❑ The “Nested Monitor Problem”
- ❑ Permits and Semaphores
- ❑ Concurrently available methods
  - ☞ Priority
  - ☞ Interception
  - ☞ Readers and WRiters
- ❑ Optimistic Methods

Sources

- ❑ D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996
- ❑ Doug Lea, *On-line Supplement to Concurrent Programming in Java,* `http://gee.cs.oswego.edu/dl/cpj/index.html`

Condition Objects

Intent

Condition objects encapsulate the waits and notifications used in guarded methods.

Applicability

- ❑ To simplify class design by off-loading waiting and notification mechanics.
  - ☞ Because of the limitations surrounding the use of condition objects in Java, in some cases the use of condition objects will *increase* rather than decrease design complexity!
- ❑ As an efficiency manoeuvre. By isolating conditions, you can often avoid notifying waiting threads that could not possibly proceed given a particular state change.
- ❑ As a means of encapsulating special scheduling policies surrounding notifications, for example to impose fairness or prioritization policies.
- ❑ In the particular cases where conditions take the form of “permits” or “latches.”

A Simple Condition Object

Condition objects implement this interface:

```
public interface Condition {
   public void await();        // wait for some condition
   public void signal();       // signal that some condition holds
}
```

Suppose we tried to encapsulate guard conditions with this class:

```
public class SimpleConditionObject implements Condition {
   public synchronized void await() {
      try { wait(); } catch (InterruptedException ex) {}
   }
   public synchronized void signal() {
      notifyAll();
   }
}
```

Careless use of this class can lead to the “Nested Monitor Problem”

The Nested Monitor problem

Avoid designs like this!

```
public class BoundedCounterVBAD implements BoundedCounter { // DO NOT USE!!!
   protected long count_ = MIN;
   protected Condition notMin_ = new SimpleConditionObject();
   protected Condition notMax_ = new SimpleConditionObject();

   public synchronized long value() { return count_; }

   public synchronized void inc() {
      while (count_ == MAX)
         notMax_.await();       // wait until count is not at max allowed value
      if (count_++ == MIN)
         notMin_.signal();      // signal that count is not at min allowed value
   }

   public synchronized void dec() {
      while (count_ == MIN)
         notMin_.await();
      if (count_-- == MAX)
         notMax_.signal();      // can't get here if locked out by inc()!!!
   }
}
```

Nested monitor lockouts occur whenever a blocked thread holds the lock for an object containing the method that would otherwise provide a notification to unblock the wait.

Solving the Nested Monitors problem

You must ensure that:

- ❑ Waits do not occur while synchronization is held on the host object.
  - ☞ This leads to a guard loop that *reverses* the synchronization seen in the faulty version.
- ❑ Notifications are never missed.
  - ☞ The entire guard wait loop should be enclosed within synchronized blocks on the condition object.
- ❑ Notifications do not deadlock.
  - ☞ All notifications should be performed only upon release of all synchronization except of that for the notified condition object.
- ❑ If the helper object maintains any state, that it is always consistent with that of the host, and if it shares any state with the host, that access is properly synchronized.

Example solution

```
public class BoundedCounterVCV implements BoundedCounter {
   protected long count_ = MIN;
   protected Condition notMin_ = new SimpleConditionObject();
   protected Condition notMax_ = new SimpleConditionObject();

   public synchronized long value() { return count_; }

   public void inc() {                    // NOT synched!
      boolean wasMin = false;             // record notification condition for later
      synchronized(notMax_) {             // synch guard loop on condition object
         for (;;) {                       // the recast guard loop
            synchronized(this) {
               if (count_ < MAX) {        // check and act
                  wasMin = (count_++ == MIN);
                  break;
               }
            }
            notMax_.await();              // release host synch before wait
         }
      }
      if (wasMin) notMin_.signal();       // release all sync before signal
   }

   public void dec() {                    // symmetric
      ...
}
```

Permits and Semaphores

Intent

Bundle synchronization in a condition object when synchronization is mainly concerned with tracking the value of a counter.

Applicability

- ❑ When any given `await` may proceed only if there have been more signals than awaits.
  - ☞ More generally, if there are enough “permits”, where every signal increments and every await decrements the number of permits.
- ❑ You need to guarantee the absence of missed signals.
  - ☞ Unlike simple condition objects, semaphores work even if one thread enters its await after another thread has signalled that it may proceed.
- ❑ The host classes using them can arrange to invoke `Condition` methods *outside* of synchronized methods or code blocks.

Permits and Semaphores — design steps

❑ Define a class implementing `Condition` that maintains a permit count, and immediately releases await if there are already enough permits.

❑ As with all kinds of condition objects, the classes using them must *avoid invoking await inside of synchronized methods and code blocks*.

☞ One way to help ensure this is to use a before/after design of the form:

```
class Host {
   Condition aCondition_;
   Condition anotherCondition_;
   Condition aThirdCondition_;

   public method m1() {
      aCondition_.await();         // not synched
      doM1();                      // synched
      for each Condition c enabled by m1()
         c.signal();               // not synched
   }

   protected synchronized doM1() { /* the actions of m1() */ }
}
```

Variants

- ❑ Permit Counters (Counting Semaphores)
  - ☞ Just keep track of the number of “permits”
  - ☞ Can use `notify` instead of `notifyAll` if class is `final`
- ❑ Fair Semaphores
  - ☞ Maintain FIFO queue of threads waiting on a `SimpleCondition`
- ❑ Locks and Latches
  - ☞ Locks can be acquired and released in separate methods
  - ☞ Keep track of thread holding the lock so locks can be reentrant!
  - ☞ A latch is set to *true* by `signal`, and always stays true

See the On-line supplement for details.

Concurrently Available Methods

Intent

Non-interfering methods comprising a service an be made concurrently available by splitting them into different objects or aspects of the same object, while tracking state and execution conditions to enable and disable the methods according to a given concurrency control policy.

Applicability

- ❑ Host objects are typically accessed across many different threads.
- ❑ Host services are not completely interdependent, so need not be performed under mutual exclusion.
- ❑ You need better throughput for one or more of the services provided by the object, and need to eliminate nonessential blocking on synchronization locks.
- ❑ You want to prevent various accidental or malicious denial of service attacks in which synchronized methods on a host block because some client forever holds its lock.
- ❑ Use of full synchronization would needlessly make host objects prone to deadlock or other liveness problems.

Concurrent Methods — design steps

Layer concurrency control policy over mechanism by:

- ❑ *Policy Definition:*
  - ☞ When may methods run concurrently?
  - ☞ What happens when a disabled method is invoked?
  - ☞ What priority is assigned to waiting tasks?
- ❑ *Instrumentation:*
  - ☞ Define state variables that can detect and enforce policy.
- ❑ *Interception:*
  - ☞ Have the host object intercept public messages and then relay them under the appropriate conditions to the methods that actually perform the actions.

Priority

- ❑ Priority may depend on any of:
  - ☞ Intrinsic attributes of the tasks (class and instance variable values).
  - ☞ Representations of task priority, cost, price, or urgency.
  - ☞ The number of tasks waiting for some condition.
  - ☞ The time at which each task is added to a queue.
  - ☞ Fairness — guarantees that each waiting task will eventually run.
  - ☞ The expected duration or time to completion of each task.
  - ☞ The desired completion time of each task.
  - ☞ Termination dependencies among tasks.
  - ☞ The number of tasks that have completed.
  - ☞ The current time.

Interception

Interception strategies include:

- ❑ Pass-Throughs
  - ☞ The host maintains a set of immutable references to helper objects and simply relays all messages to them within unsynchronized methods.
- ❑ Lock-Splitting
  - ☞ Instead of splitting the class, split the synchronization locks associated with subsets of functionality
- ❑ Before/After methods
  - ☞ Public methods contain before/after processing surrounding calls to non-public methods in the host that perform the services.

Concurrent Reader and Writers

"Readers and Writers" is a family of concurrency control designs that provide various policies governing concurrent invocation of *non-mutating accessors* ("Readers") and *mutative, state-changing operations* ("Writers").

The basic idea is to let any number of readers to concurrently execute as long as there are no writers, but writers have exclusive access.

Individual policies must address:

- ❑ Can new Readers join already active Readers even if a Writer is waiting?
  - ☞ If yes, Writers may starve; if not, the throughput of Readers decreases.
- ❑ If both Readers and Writers are waiting for a Writer to finish, which should you let in first?
  - ☞ Readers? A Writer? Earliest first? Random? Alternate?
  - ☞ Similar choices are available after termination of Readers.
- ❑ Can Readers upgrade to Writers without having to give up access?

Before/after methods are the simplest way to implement Readers and Writers policies.

Readers and Writers example

The following example illustrates a common set of choices:

- ❑ Block incoming Readers if there are waiting Writers.
- ❑ “Randomly” choose among incoming threads. (I.e., leave the choice to the native Java scheduler)
- ❑ No upgrade mechanisms.

```
public abstract class RWVT {
   protected int activeReaders_ = 0;     // threads executing read_
   protected int activeWriters_ = 0;     // always zero or one
   protected int waitingReaders_ = 0;    // threads not yet in read_
   protected int waitingWriters_ = 0;    // same for write_

   protected abstract void read_();      // implement in subclasses
   protected abstract void write_();

   public void read() {   beforeRead();    read_();    afterRead(); }

   public void write() {  beforeWrite();   write_();   afterWrite(); }

   protected boolean allowReader() {
      return waitingWriters_ == 0 && activeWriters_ == 0;
   }
```

```
    protected boolean allowWriter() {
       return activeReaders_ == 0 && activeWriters_ == 0;
    }
    protected synchronized void beforeRead() {
       ++waitingReaders_;
       while (!allowReader())
          try { wait(); } catch (InterruptedException ex) {}
       --waitingReaders_;
       ++activeReaders_;
    }
    protected synchronized void afterRead() {
       --activeReaders_; notifyAll();
    }
    protected synchronized void beforeWrite() {
       ++waitingWriters_;
       while (!allowWriter())
          try { wait(); } catch (InterruptedException ex) {}
       --waitingWriters_;
       ++activeWriters_;
    }
    protected synchronized void afterWrite() {
       --activeWriters_; notifyAll();
    }
 }
```

Optimistic Methods

Intent

Optimistic methods attempt actions, but rollback state if the actions could have been interfered with by the actions of other threads. After rollback, they either throw failure exceptions or retry the actions.

Applicability

- ❑ Clients can tolerate either failure or retries.
  - ☞ If not, consider using guarded methods .
- ❑ You can avoid or cope with livelock.
- ❑ You have a way to deal with actions occurring before failure detection
  - ☞ *Provisional action:* "pretend" to act, delaying commitment of effects until the possibility of failure has been ruled out.
  - ☞ *Rollback/Recovery:* undo the effects of each performed action. If messages are sent to other objects, they must be undone with "anti-messages"

Optimistic Methods — design steps

- ❑ Collect and encapsulate all mutable state so that it can be tracked as a unit.
  - ☞ Define an immutable helper class holding values of all instance variables.
  - ☞ Define a representation class, but make it mutable (allow instance variables to change), and additionally include a version number (or transaction identifier) field or even a sufficiently precise time stamp.
  - ☞ Embed all instance variables, plus a version number, in the host class, but define `commit` to take as arguments all assumed values and all new values of these variables.
  - ☞ Maintain a serialized copy of object state.
  - ☞ Various mixtures of the above ...

Detect failure ...

- Provide an operation that simultaneously detects version conflicts and performs updates via a method of the form:

```
class Optimistic {                       // generic code sketch
   private State currentState_;          // State is any type

   synchronized boolean commit(State assumed, State next) {
      boolean success = (currentState_ == assumed);
         if (success)
         currentState_ = next;
      return success;
   }
}
```

- Structure the main actions of each public method as

```
State assumed = currentState();
State next = ...
if (!commit(assumed, next))
   rollback();
else
   otherActionsDependingOnNewStateButNotChangingIt();
```

Handle conflicts ...

- ❑ Choose and implement a policy for dealing with commitment failure:
  - ☞ Throw an exception upon commit failure that tells a client that it may retry. (Of course, this kicks the decision back to the caller, which may or may not be in a better position to decide whether to retry.)
  - ☞ Internally retry the action until it succeeds.
  - ☞ Retry some bounded number of times, or until a timeout occurs, finally throwing an exception.
  - ☞ Synchronize the method, precluding commit failure. This can be done even when other methods in the class use exceptions or retries.
- ❑ Take precautions to ensure that retries are based upon accurate, current values of instance variables.
  - ☞ If state is maintained in an immutable helper object accessed via a single reference in the class, then this reference should be declared `volatile`. All accessor methods can be left as unsynchronized.

`volatile` *specifies that a variable changes asynchronously and the compiler should not attempt optimizations with it (such as using a copy stored in a register).*

Ensure progress ...

- Take precautions to ensure progress in case of internal retries within *state-dependent* methods.
  - ☞ Optimistic state-dependent methods require use of a busy-wait spin loop in which it is counterproductive to immediately retry the method.
  - ☞ Yielding may not be effective unless all threads have reasonable priorities and the Java scheduler at least approximates fair choice among waiting tasks (which it is not guaranteed to do)!

- Limit retries.
  - ☞ Unless there is some independent assurance that the method will eventually succeed, retries can result in livelock.

An optimistic Bounded Counter

```
public class BoundedCounterVOPT implements BoundedCounter {
   protected volatile Long count_ = new Long(MIN);

   protected synchronized boolean commit(Long oldc, Long newc) {
      boolean success = (count_ == oldc);
      if (success) count_ = newc;
      return success;
   }

   public long value() { return count_.longValue(); }

   public void inc() {
      for (;;) {                              // thinly disguised busy-wait!
         Long c = count_; long v = c.longValue();
         if (v < MAX && commit(c, new Long(v+1))) break;
         Thread.currentThread().yield();      // is there another thread?!
      }
   }

   public void dec() {
      for (;;) {
         Long c = count_; long v = c.longValue();
         if (v > MIN && commit(c, new Long(v-1))) break;
         Thread.currentThread().yield();
      }
   }
}
```

Summary

You Should Know The Answers To These Questions:

- ❑ What are “condition objects”? How can they make your life easier? Harder?
- ❑ What is the “nested monitor problem”? How can you avoid it?
- ❑ What are “permits” and “semaphores”? When is it natural to use them?
- ❑ Why (when) can semaphores use `notify()` instead of `notifyAll()`?
- ❑ When should you consider allowing methods to be concurrently available?
- ❑ What kinds of policies can apply to concurrent Readers and Writers?
- ❑ How do optimistic methods differ from guarded methods?

Can You Answer The Following Questions?

- ✎ *What is the easiest way to avoid the nested monitor problem?*
- ✎ *What assumptions do nested monitors violate?*
- ✎ *How can the obvious implementation of semaphores (in Java) violate fairness?*
- ✎ *How does “partial synchronization” differ from “concurrently available methods”?*
- ✎ *When should you prefer optimistic methods to guarded methods?*

6. Architectural Styles for Concurrency

Overview

- What is Software Architecture?
- Three-layered application architecture
- Flow architectures
- Blackboard architectures

Sources

- M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
- F. Buschmann, et al., *Pattern-Oriented Software Architecture — A System of Patterns*, John Wiley, 1996.
- D. Lea, *Concurrent Programming in Java — Design principles and Patterns*, The Java Series, Addison-Wesley, 1996.
- N. Carriero and D. Gelernter, *How to Write Parallel Programs: a First Course*, MIT Press, Cambridge, 1990.

Software Architecture

A Software Architecture defines a system in terms of computational components and interactions amongst those components.

An Architectural Style defines a family of systems in terms of a pattern of structural organization.

— cf. Shaw & Garlan, Software Architecture, pp. 3, 19

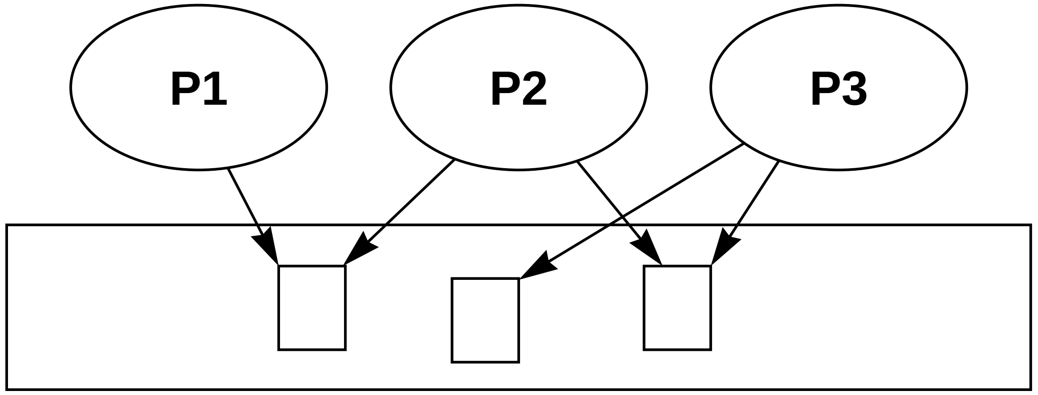
Architectural style

Architectural styles typically entail four kinds of properties:

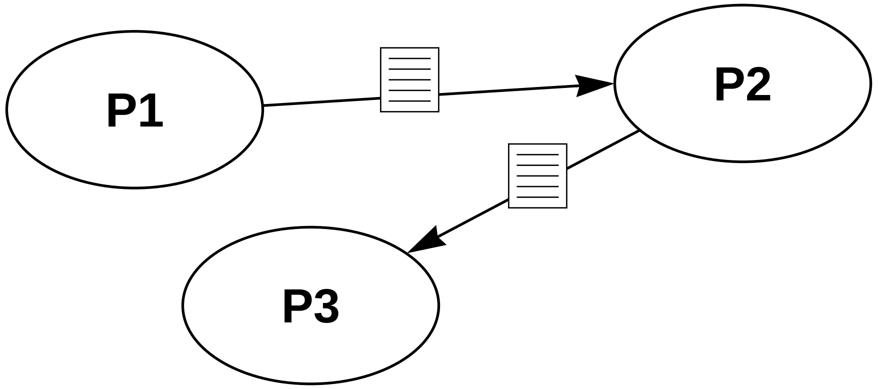
- ❑ A *vocabulary* of design elements
  - ☞ e.g., “pipes”, “filters”, “sources”, and “sinks”
- ❑ A set of *configuration rules* that constrain compositions
  - ☞ e.g., pipes and filters must alternate in a linear sequence
- ❑ A *semantic interpretation*
  - ☞ e.g., each filter reads bytes from its input stream and writes bytes to its output stream
- ❑ A set of *analyses* that can be performed
  - ☞ e.g., if filters are “well-behaved”, no deadlock can occur, and all filters can progress in tandem

Communication Styles

Shared Variables:

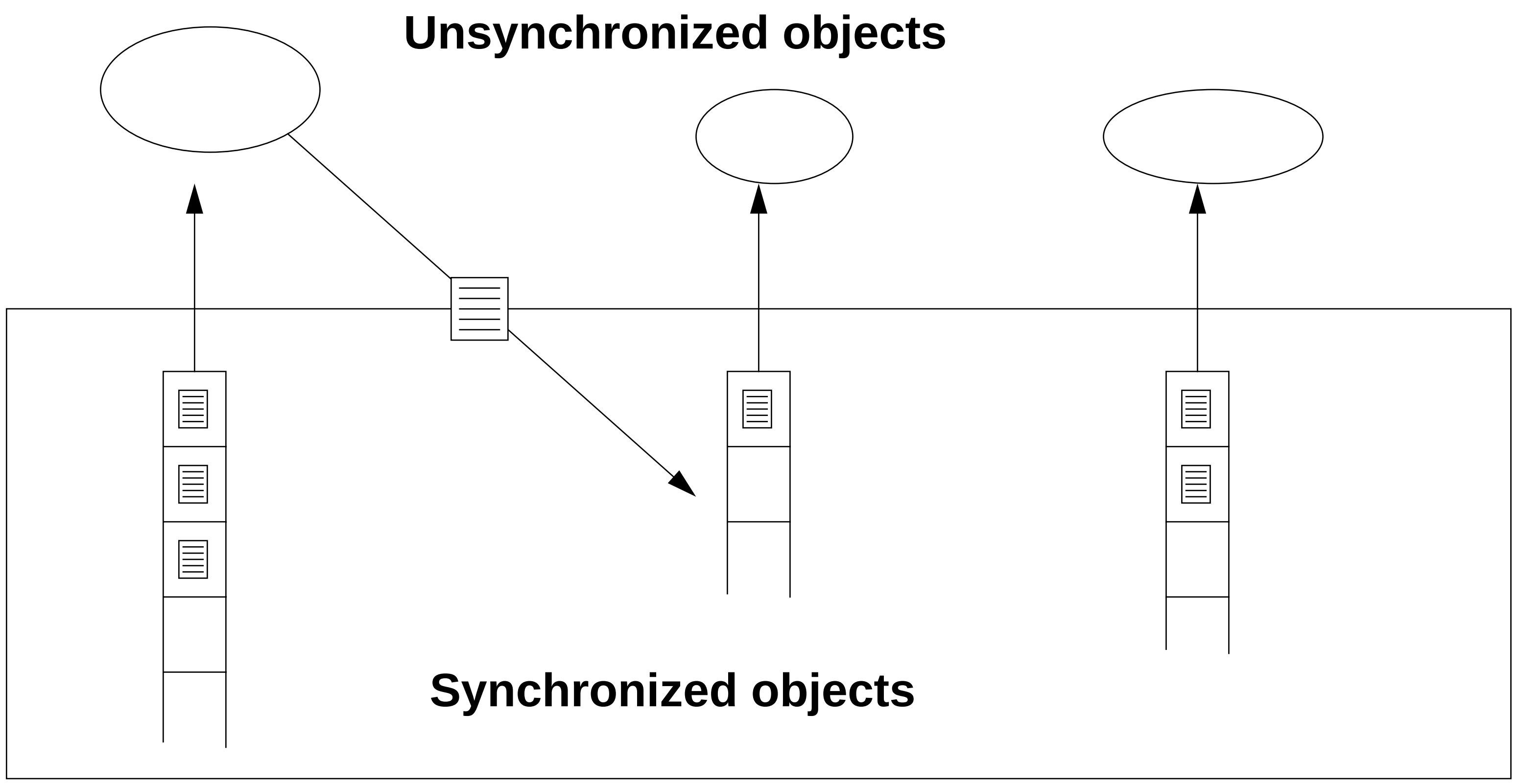


Message-Passing:



Simulated Message-Passing

Most concurrency and communication styles can be simulated by one another:



Message-passing can be modelled by associating message queues to each process.

Three-layered Application Architectures

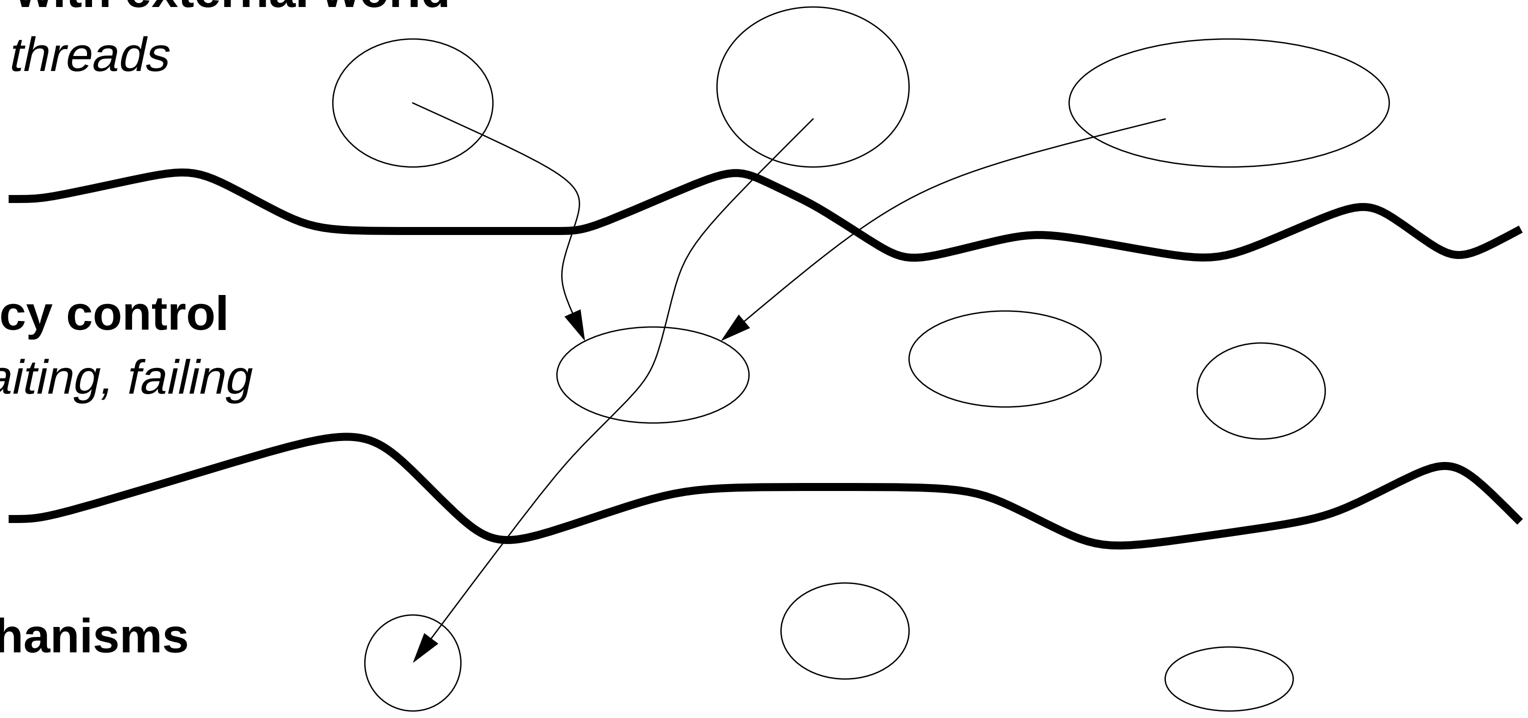
Interaction with external world

Generating threads

Concurrency control

Locking, waiting, failing

Basic mechanisms



This kind of architecture avoids nested monitor problems by restricting concurrency control to a single layer.

Problems with Layered Designs

Hard to extend beyond three layers because:

- ❑ Control is restricted to before/after — not within
- ❑ Control may depend on unavailable information
  - ☞ Because it is not safely accessible
  - ☞ Because it is not represented (e.g., message history)
- ❑ Actions in control code may encounter conflicting policies
  - ☞ E.g., nested monitor lockouts
- ❑ Ground actions may need to know current policy
  - ☞ E.g., blocking vs. failing

Partial solutions:

- ❑ Explicit policy compatibility constraints
- ❑ Explicit nesting constraints
- ❑ Delegated control

Flow Architectures

Many synchronization problems can be avoided by arranging things so that information only flows in one direction from sources to filters to sinks:

- ❑ Unix “pipes and filters”:
  - ☞ Processes are connected in a linear sequence
- ❑ Control systems:
  - ☞ events are picked up by sensors, processed, and generate new events
- ❑ Workflow systems
  - ☞ Electronic documents flow through workflow procedures

Flow Stages

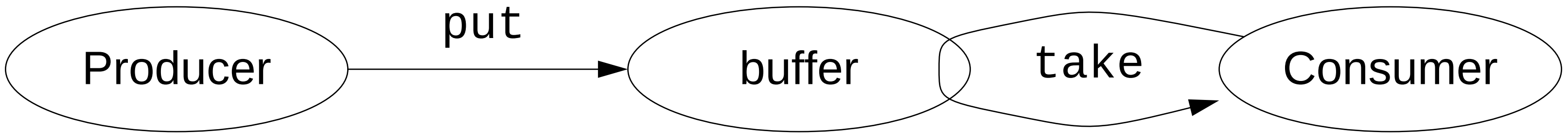
Every flow stage is a producer or consumer or both:

- *Splitters* (forks) have multiple successors
  - ☞ *Multicasters* clone results to multiple consumers
  - ☞ *Routers* divide results amongst consumers
- *Mergers* have multiple predecessors
  - ☞ *Collectors* (Multiplexers) interleave inputs to a single consumer
  - ☞ *Combiners* process multiple input to produce a single result
- *Conduits* have both multiple predecessors and consumers

Flow Policies

Flow can be pull-based, push-based, or a mixture:

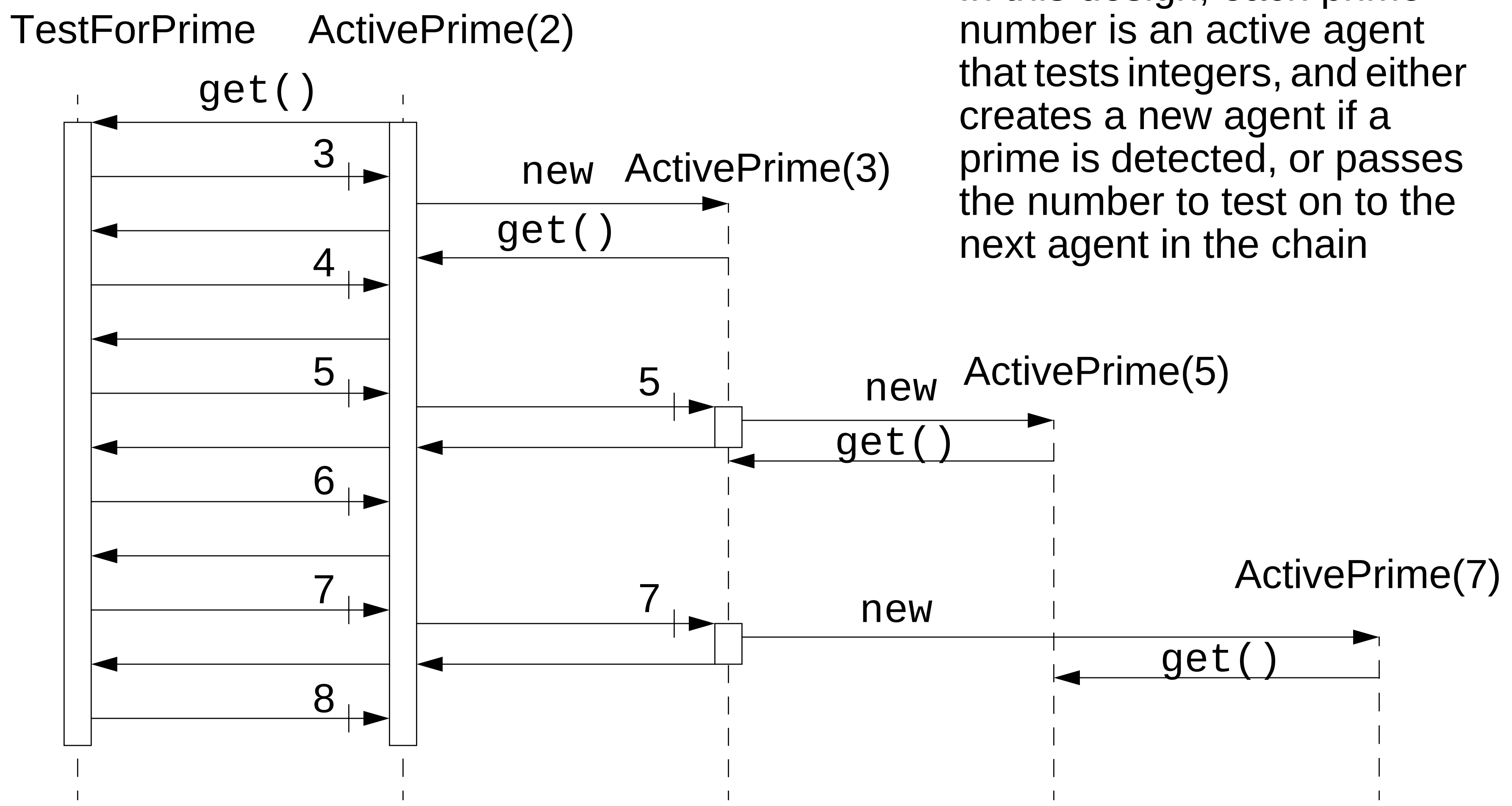
- Pull-based flow: Consumers *take* results from Producers
- Push-based flow: Producers *put* results to Consumers
- Buffers:
  - Put-only buffers (*relays*) connect push-based stages
  - Take-only buffers (*pre-fetch buffers*) connect pull-based stages
  - Put-Take buffers connect push-based stages to pull-based stages



Limiting Flow

- Unbounded buffers:
  - ☞ If producers are faster than consumers, buffers may exhaust available memory
- Unbounded threads:
  - ☞ Having too many threads can exhaust system resources more quickly than unbounded buffers
- Bounded buffers:
  - ☞ Tend to be either always full or always empty, depending on relative speed of producers and consumers
- Bounded thread pools:
  - ☞ Harder to manage than bounded buffers

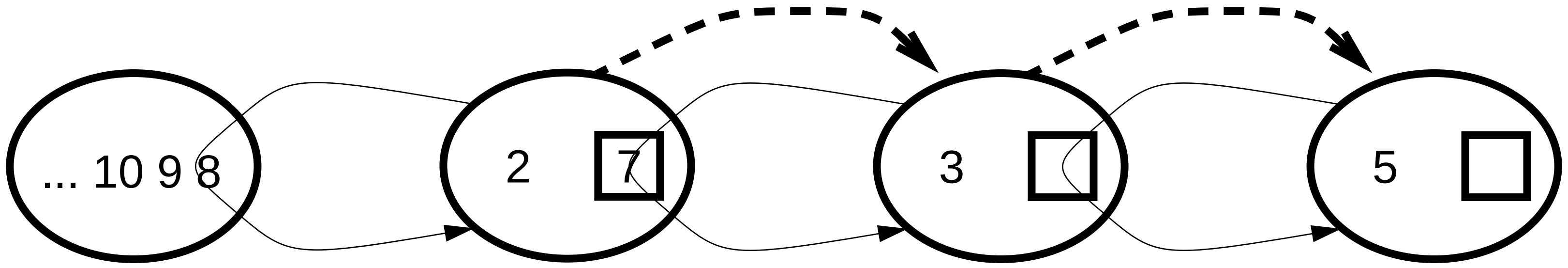
Example: a Pull-based Prime Sieve



In this design, each prime number is an active agent that tests integers, and either creates a new agent if a prime is detected, or passes the number to test on to the next agent in the chain

Using Put-Take Buffers

Each ActivePrime will use a one-slot buffer to feed values to the next ActivePrime:



Initially we create an ActivePrime for the value 2, connected to a TestForPrime generator:

```
public class PrimeSieve {
   public static void main(String args[]) {
      genPrimes(1000);
   }
   public static void genPrimes(int n) {
      try {
         ActivePrime firstPrime = new ActivePrime(2, new TestForPrime(n));
      } catch (Exception e) { }
```

```
          }
      }
```

Pull-based integer sources

Active primes get numbers to test from an `IntSource` interface:

```
interface IntSource {
   int getInt();
}

class TestForPrime implements IntSource {
   private int nextValue;
   private int maxValue;
   public TestForPrime(int max) {
      this.nextValue = 3;
      this.maxValue = max;
   }
   public int getInt() {                  // No synchronization need
      if (nextValue < maxValue) { return nextValue++; }
      else { return 0; }
   }
}
```

A Put-Take Buffer

```
class Slot {                                        // a one-slot buffer
   private Object slotVal;                          // initially null
   public synchronized void put(Object val) {      // This is the only
      while (slotVal != null) {                     // synchronized object
         try { wait(); }
         catch (InterruptedException e) { }
      }
      slotVal = val;
      notifyAll();                                  // same as notify(),
      return;                                       // if only one producer
   }                                                // and one consumer
   public synchronized Object get() {
      Object rval;
      while (slotVal == null) {
         try { wait(); }
         catch (InterruptedException e) { }
      }
      rval = slotVal;
      slotVal = null;
      notifyAll();
      return rval;
   }
}
```

The ActivePrime Class

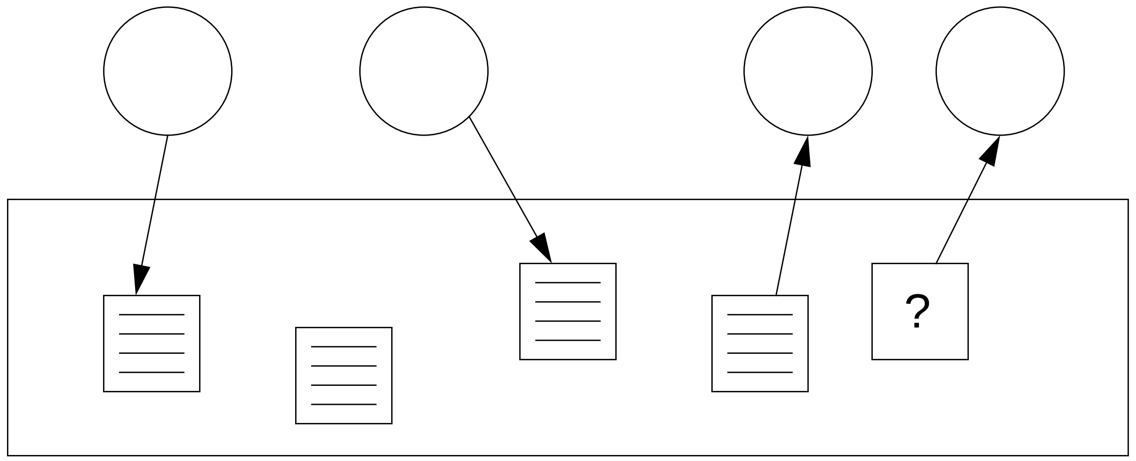
```
class ActivePrime extends Thread implements IntSource {
   private static IntSource lastPrime;  // where to link the next prime
   private int value;                   // value of this prime
   private int square;                  // square of this prime
   private IntSource intSrc;            // source of ints to test
   private Slot slot;                   // to pass test value to next ActivePrime
   public ActivePrime(int value, IntSource intSrc) throws ActivePrimeFailure
   {
      this.value = value;
      this.square = value*value;
      this.intSrc = intSrc;
      slot = new Slot();                // NB: private
      lastPrime = this;                 // NB: set class variable
      System.out.print(value + " ");
      System.out.flush();
      this.start();                     // become active
   }
   public int value() { return this.value; }
   ...
}
```

ActivePrime ...

```
class ActivePrime extends Thread implements IntSource {
   ...
   private void putInt(int val) { slot.put(new Integer(val)); }
   public int getInt() {
      int rval;
      rval = ((Integer) slot.get()).intValue();     // may block
      return rval;
   }
   public void run() {
      int testValue = intSrc.getInt();               // may block
      while (testValue != 0) {
         if (testValue < this.square) {
            try { new ActivePrime(testValue, lastPrime); }
            catch (Exception e) { testValue = 0; } // stop the thread
         } else if ((testValue % this.value) > 0) {
            this.putInt(testValue);                  // may block
         }
         testValue = intSrc.getInt();               // may block
      }
      putInt(0);                                     // stop condition
   }
}
```

Blackboard Architectures

Blackboard architectures put all synchronization in a “coordination medium” where agents can exchange messages.

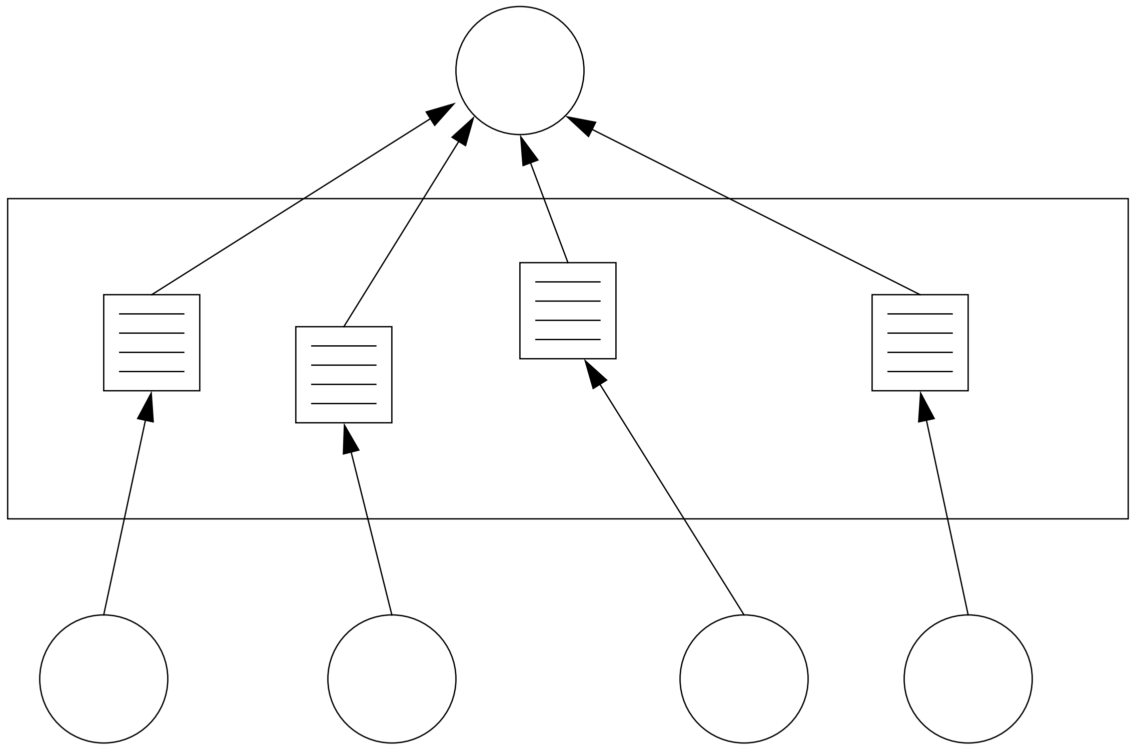


Agents do not exchange messages directly, but post messages to the blackboard, and retrieve messages either by reading from a specific location (i.e., a *channel*), or by posing a query (i.e., a *pattern* to match).

Linda is a “coordination language” that provides primitives for implementing blackboard architectures ...

Result Parallelism

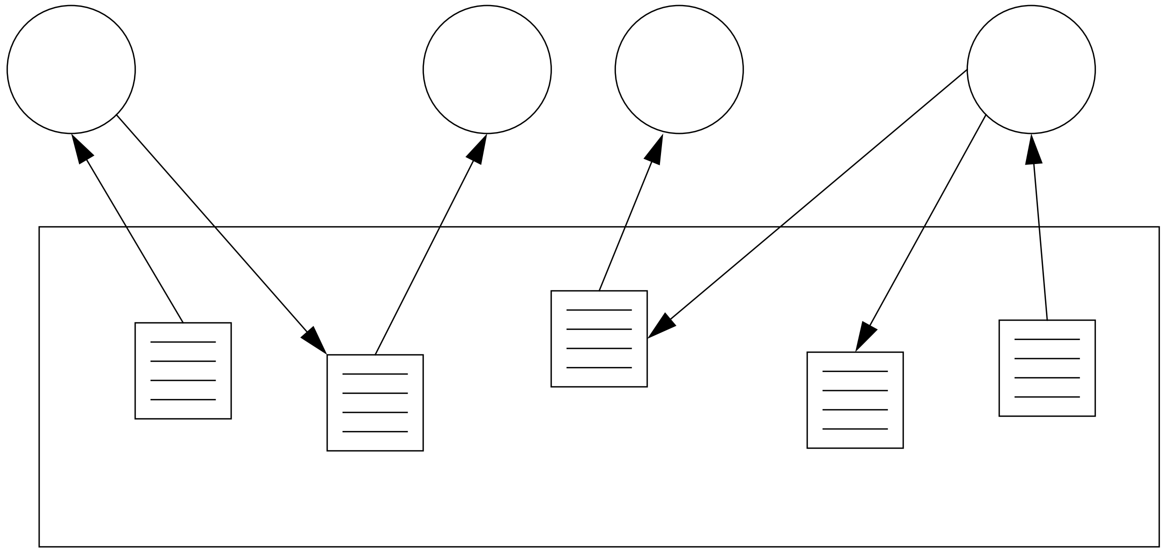
Result parallelism is a blackboard architectural style in which workers are spawned to produce each part of a more complex problem.



Workers may be arranged hierarchically ...

Agenda Parallelism

Agenda parallelism is a blackboard style in which workers retrieve tasks to perform from a blackboard, and may generate new tasks to perform.

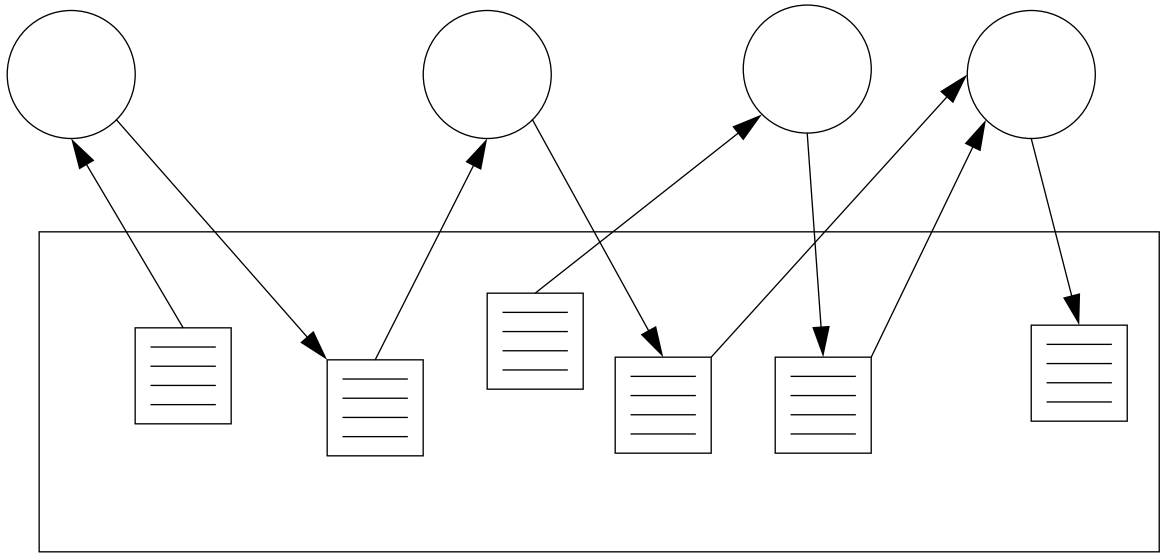


Workers repeatedly retrieve tasks until everything is done.

Workers are typically able to perform arbitrary tasks.

Specialist Parallelism

Specialist parallelism is a style in which each workers is *specialized* to perform a particular task.



Specialist designs are equivalent to message-passing, and are generally organized as flow architectures, with each specialist producing results for the next specialist to consume.

Summary

You Should Know The Answers To These Questions:

- ❑ What is a Software Architecture?
- ❑ What are advantages and disadvantages of Layered Architectures?
- ❑ What is a Flow Architecture? What are the options of tradeoffs?
- ❑ What are Blackboard Architectures? What are the options and tradeoffs?

Can You Answer The Following Questions?

- ✎ *How would you model message-passing agents in Java?*
- ✎ *How would you classify Client/Server architectures?*
  Are there other useful styles we haven't yet discussed?
- ✎ *How can we prove that the Active Prime Sieve is correct? Are you sure that new Active Primes will join the chain in the correct order?*
- ✎ *Which Blackboard styles are better when we have multiple processors?*
  Which are better when we just have threads on a monoprocessor?

7. Coordination Models and Languages

Overview

- ❑ Coordinated Systems
- ❑ Coordination Languages
- ❑ Coordination Models
- ❑ Blackboard Coordination Models
  - ☞ Linda
  - ☞ JavaSpace - Jada
- ❑ Multiset Coordination Models
  - ☞ GAMMA
- ❑ Object Oriented Coordination Languages
  - ☞ FLO/C
  - ☞ ATOM
- ❑ SCG Coordination Research

Sources

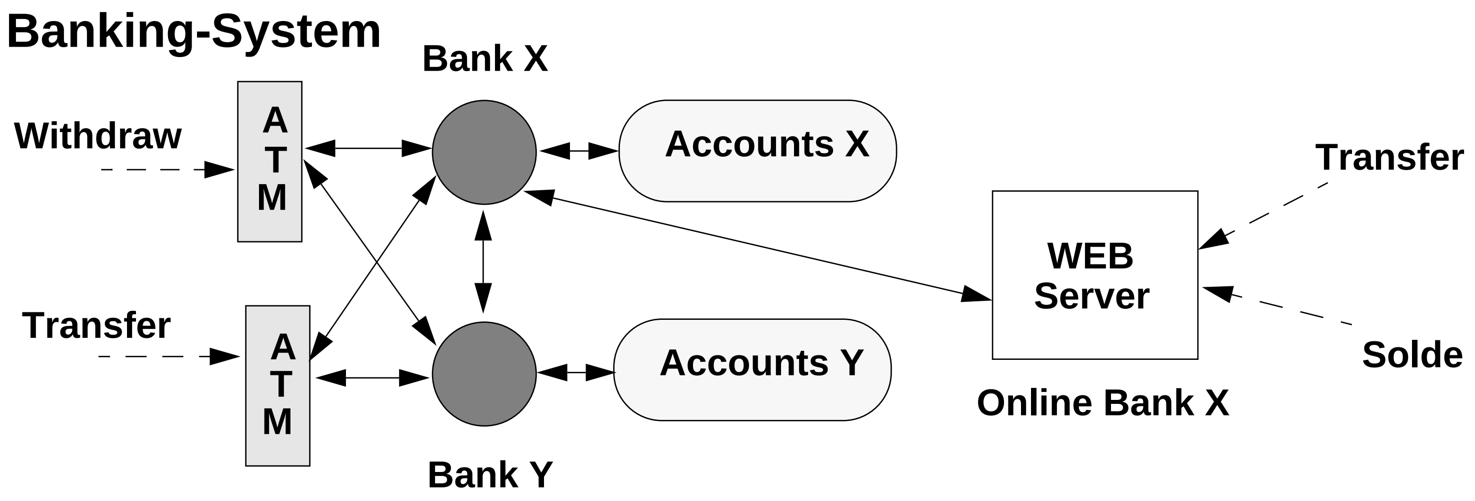
- ❑ N.Carriero and D.Gelernter, *How to Write Parallel Programs*, MIT Press, 1991
- ❑ P. Ciancarini, *Tutorial: Coordination and Software Enginnering*, 1997

Juan Carlos Cruz, cruz@iam.unibe.ch, http://www.iam.unibe.ch/~cruz

Coordinated Systems

Modern software systems are systems composed of encapsulated software entities (i.e. active objects, agents, actors, etc.) that run asynchronously and process information concurrently.

— cf. Hewitt "Offices are Open Systems", pp 270-287



Software entities that compose those systems cooperate in complex ways in order to produce results. It is also because of they cooperate, that they may need to coordinate their actions.

Why they may need to coordinate actions?

- ❑ No entity has enough competence to solve the entire problem
  - ☞ ATMs don’t know how to make Withdraws or Transfers on accounts, different banks can apply different policies (i.e. taxes, etc.)
- ❑ No entity has enough resources to solve the entire problem
  - ☞ ATMs don’t not have infinite disk-space to keep the information of all the accounts in the system
- ❑ No entity has enough information to solve the entire problem
  - ☞ ATMs and Online Servers do not have the account’s information to realize the banking operations (i.e they may need to verify soldes, etc.)
- ❑ There are dependencies between the activities they execute
  - ☞ Concurrent banking operations may modify a same account at the same time
- ❑ There are some global constraints they have to respect
  - ☞ All ATM may realize banking operations on whatever bank in the system

Managing Coordination is a key aspect in the development of modern software systems.
— *cf. T.Malone“The Implications of The Digital Age”, Int. Comp., V1 N3, May 97, pp 8-20*

What is Coordination?

Coordination concerns the organisation in time and in space of the behaviour of a group of entities in order to either improve their collective results, or to reduce their conflicts.

— cf. Cruz et al. “A Coordination Component Framework for Open Systems”

From the viewpoint of the Software Engineering Coordination can be defined as:

Coordination refers to the process of building programs by gluing together active pieces.

— cf. Carriero & Gelernter, How to Write Parallel Programs, pp. 8

An active piece may be a process, task, thread or any other locus of execution that executes concurrently and asynchronously with the rest.

Coordination Language and Model

A Coordination Language provides the "glue" that binds separate active pieces into software systems. The "glue "must allow these independent pieces to ***communicate and to synchronize*** *with each other exactly as the need to.*

— cf. Carriero & Gelernter, How to Write Parallel Programs, pp. 8

All Coordination Language embodies a Coordination Model.

A Coordination Model is an abstract (semantic) framework useful to study and understand coordination problems when designing coordinated systems.

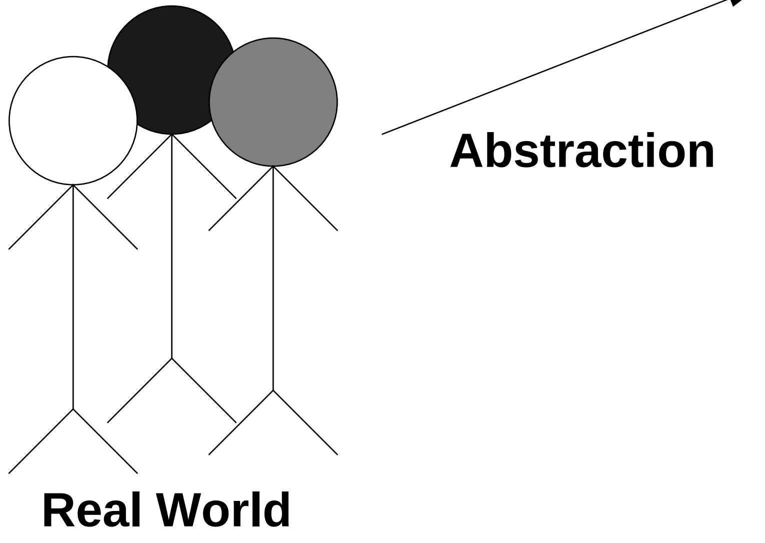
— cf. Ciancarini, "Coordination and Software Engineering", pp. 4

A coordination model defines how active pieces (i.e agents) interact and how their interactions can be controlled. This includes:

Creation and destruction of agents, control of communication flows among agents, control of spatial distribution and mobility of agents as well as synchronization and distribution of actions over time. — cf. Ciancarini,

Model vs. Language

A model is an abstraction of something for the purpose of understanding and studying it. Because a model omits nonessential details, it is easier to manipulate than the original entity. Abstraction is a fundamental human capability that permits us to deal with complexity.



Person
Name
Age
Color
Nationality

Model

“A particular way of thinking about a problem”

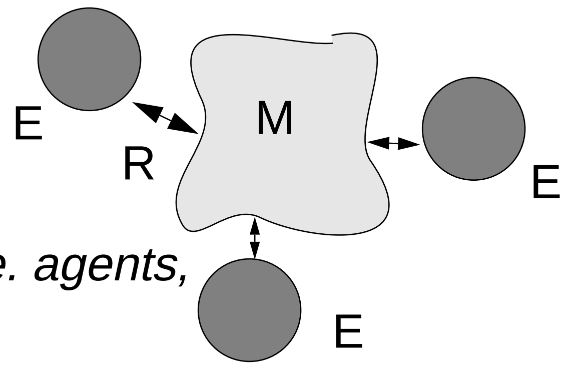
```
Class Person {
   String name;
   int age;
   Color color;
   String Nationality
}
```

Computer Language

“A way to realize or implement a model in a computer”

Coordination Models

*A Coordination Model is a triple (**E**,**M**,**L**), where:*



*. **E** are the coordinable entities (components):*

These are the active agents which are coordinated (i.e. agents, processes, active objects, tuples, atoms, etc.).

*. **M** are the coordinating media (connectors):*

These are the coordinators of interagent entities (i.e. channels, shared variables, tuple spaces, bags, etc.)

*. **L** are the coordination laws:*

They rule actions of coordinable entities (i.e. associative access, guards, synchr. constraints)

— cf. Ciancarini, "Coordination and Software Engineering", pp. 7

Coordination Language and Model

*A Coordination Language provides the “glue” that binds separate active pieces into software systems. The “glue “must allow these independent pieces to **communicate and to synchronize** with each other exactly as the need to.*

— cf. Carriero & Gelernter, How to Write Parallel Programs, pp. 8

All Coordination Language embodies a Coordination Model.

A Coordination Model is an abstract (semantic) framework useful to study and understand coordination problems when designing coordinated systems.

— cf. Ciancarini, “Coordination and Software Engineering”, pp. 4

A coordination model defines how active pieces (i.e agents) interact and how their interactions can be controlled. This includes:

Creation and destruction of agents, control of communication flows among agents, control of spatial distribution and mobility of agents as well as synchronization and distribution of actions over time. — cf. Ciancarini,

Programming Model

We can build a complete programming model out of two separate pieces - the computational model and the coordination model.

— cf. Carriero & Gelernter "Coord. Lang. and their Significance, pp. 97

The computational model allows programmers to build single computational activities

The coordination model is the glue that binds separate activities into an assemble.

Programs = Coordination + Computation

Advantages:

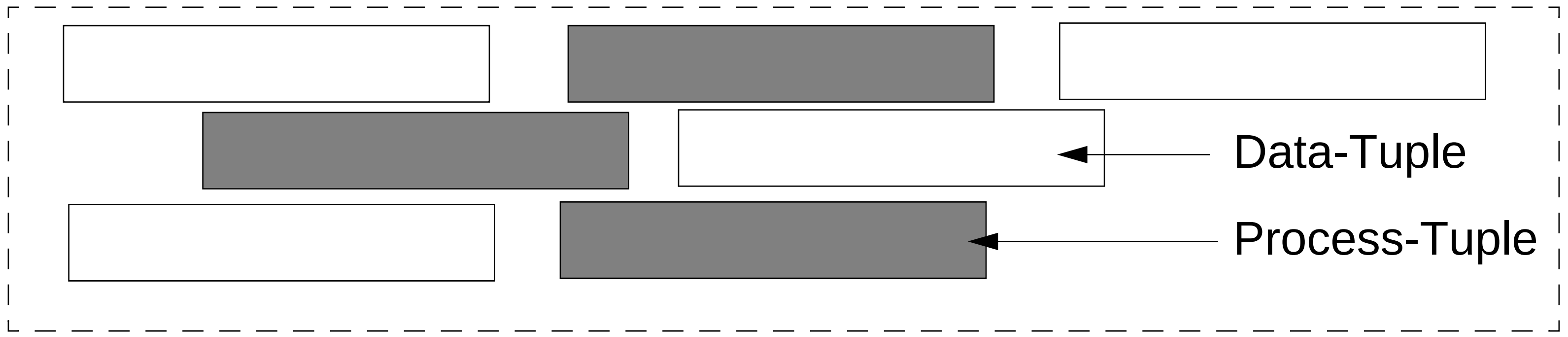
- ❏ Separation of concerns
- ❏ Reusability of the separate pieces

Linda Coordination Model

The **Linda** model is a blackboard model. Linda blackboard (called tuple space) consists of a collection of logical tuples.

☞ (“a string”, 15.01, x), (1, 0, “Hello”)

Tuple Space



There are two kinds of tuples:

- ❑ Process-Tuples which are under active evaluation.
- ❑ Data-Tuples which are passive.

Process-Tuples execute simultaneously. They exchange data by <u>generating</u>, <u>reading</u> and <u>consuming</u> Data-Tuples. A Process-Tuple that is finished executing turns into a Data-Tuple.

LInda Operations

- ❑ **out(t)** causes a tuple **t** to be added to tuple space; the executing process continues immediately.
- ❑ **in(s)** causes some tuple **t** that matches anti-tuple **s** to be <u>withdrawn</u> from tuple space. Once in(s) has found a matching tuple, the values of the actuals in **t** are assigned to the corresponding formals in **s** (i.e actual-to-formal assignment).

An anti-tuple is a tuple with typed fields; some are values (or "actuals"), others are typed place-holders (or "formals"). A formal is prefixed with a "?" marker.

☞ ("a string", ?f, ?i, x)

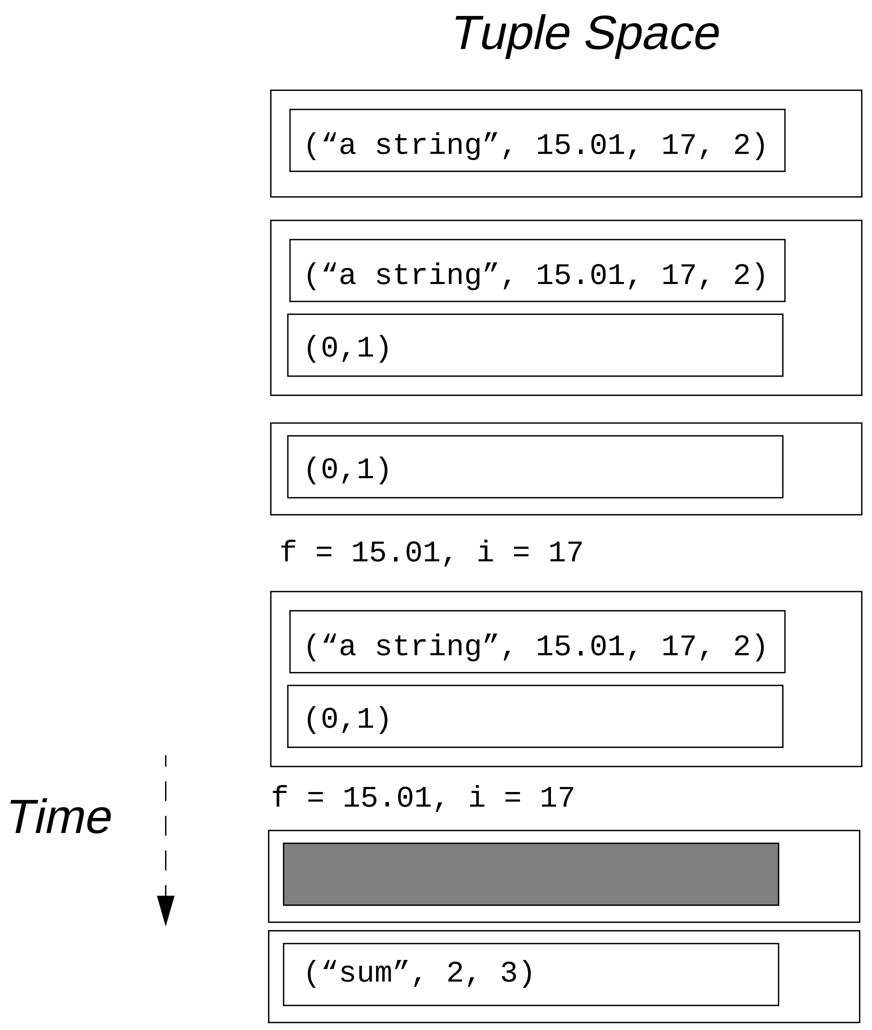
- ❑ **rd(s)** is the same as in(s), except that the matched tuple <u>remains</u> in tuple space.

If not matching t is available when in(s) or rd(s) executes, the executing process <u>suspends</u> (i.e. blocks) until one is, then proceeds as before.

- ❑ **eval(t)** is the same as out(t), except that **t** is <u>evaluated after</u> rather than before it enters tuple space.
- ❑ **inp(s)**, **rdp(s)**, attempt to locate a matching tuple and return 0 if they fail; otherwise they return 1, and perform actual-to-formal assignment.

Examples

```
1) x = 2
   out(“a string”, 15.01, 17, x)
2) out(0,1)



3) y = 2
   in(“a string”, ?f, ?i, y)



4) y = 2
   rd(“a string”, ?f, ?i, y)



5) eval(“sum”,y, f(y))
   f = Σi 1<=i<=y
```


Tuple Space
(“a string”, 15.01, 17, 2)
(“a string”, 15.01, 17, 2)
(0,1)
(0,1)
f = 15.01, i = 17
(“a string”, 15.01, 17, 2)
(0,1)
f = 15.01, i = 17
(“sum”, 2, 3)
Time

Example: Fibonacci

How to calculate the N-th fibonacci number?

```
fibonacci(n):/* fib(n+2)=fib(n)+fib(n+1) */

   if (rdp(“fibonacci”,n-1,?fibn_1)==0)
      eval(“fibonacci”,n-1,fibonacci(n-1))

   rd(“fibonacci”,n-1,?fibn_1)
   rd(“fibonacci”,n-2,?fibn_2)
   out(“fibonacci”,n, fibn_2+fibn_1)

out(“fibonacci”, 0, 0)
out(“fibonacci”, 1, 1)

eval(“fibonacci”,5, fibonacci(5))
in(“fibonacci”, 5, ?fib)
fib = ??
```

JavaSpace- (Java + Linda)

The JavaSpace package provides a distributed persistence and object exchange mechanism for code written in the Java programming language.

JADA language (Java+Linda): An implementation of the JavaSpace Specification

— P. Ciancarini and D.Rossi, "Esprit Project Page Space " Project #20197

```
import jada.Tuple;
import jada.ObjectSpace;
import java.lang.*;

class CalculFibo implements Runnable {
  ObjectSpace tpspc;
  int nfib = 0;

   public CalculFibo(ObjectSpace tp, int n) {
    tpspc = tp;
    nfib  = n;
  }
```

```
public void run() {

 int f1, f2;

 Tuple fib2 = (Tuple) tpspc.read_nb(new Tuple("fibo",
                                     new Integer(nfib-2),
                                     Tuple.IntegerClass()));
 if (fib2 == null) {
   new Thread( new CalculFibo(tpspc, nfib-2)).start();
   fib2 = (Tuple) tpspc.read(new Tuple("fibo", new Integer(nfib-2),
                                Tuple.IntegerClass()));
 }

 f2 = ((Integer)fib2.getItem(2)).intValue();
```

```
      Tuple fib1 = (Tuple) tpspc.read_nb(new Tuple("fibo",
                                           new Integer(nfib-1),
                                           Tuple.IntegerClass()));
      if (fib1 == null) {
        new Thread( new CalculFibo(tpspc, nfib-1)).start();
        fib1 = (Tuple) tpspc.read(new Tuple("fibo", new Integer(nfib-1),
                                Tuple.IntegerClass()));
      }
      f1 = ((Integer)fib1.getItem(2)).intValue();

      tpspc.out( new Tuple("fibo", new Integer(nfib), new Integer(f1+f2)));

    }
  }
```

Multiset Rewriting-The Gamma Model

Gamma is a coordination model whose main data structure is a multiset (or bag) and whose unique control structure is the Γ operator.

— cf.Banatre & Le Metayer “Programming by Multiset Transf”, CACM’93, pp.98

```
Γ((R1,A1),...,(Rm,Am))(M) =
        if ∀i ε [1,m], ∀x1,...,xn not Ri(x1,...,xn)
        then M
        else let x1,...,xn ε M, let i ε [1,m] such that Ri(x1,..xn) in
        Γ((R1,A1),...,(Rm,Am)) ((M-{x1,...,xn}) + Ai(x1,...,xn))
```

- {...} represents multisets
- (Ri,Ai) are pairs of closed functions specifying reactions

The effect of (Ri,Ai) on multiset M is to replace in M a subset of elements {x1,..,xn} such that Ri(x1,..,xn) is true by the elements of Ai(x1,...,xn).

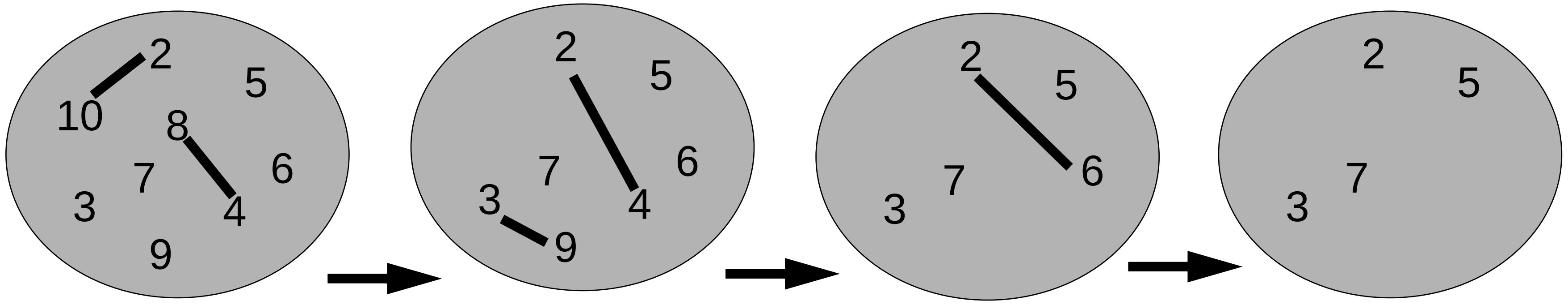
- Γ is a fixpoint operator: reactions continue until no new reaction is possible

Examples:

1) `prime_numbers(N) = Γ ((R,A))`

```
({2,..,N}) where
R(x,y) = multiple(x,y)
A(x,y) = {y}
```

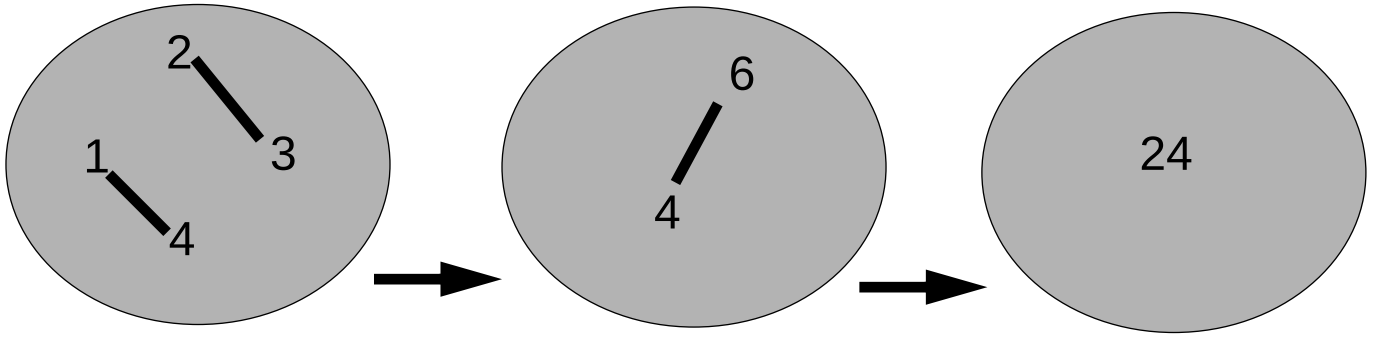
prime_numbers(10)



2) `fact(n)= G ((R,A)) ({1,...,n}) where`

```
R(x,y) = true
A(x,y) = {x*y}
```

fact(4)

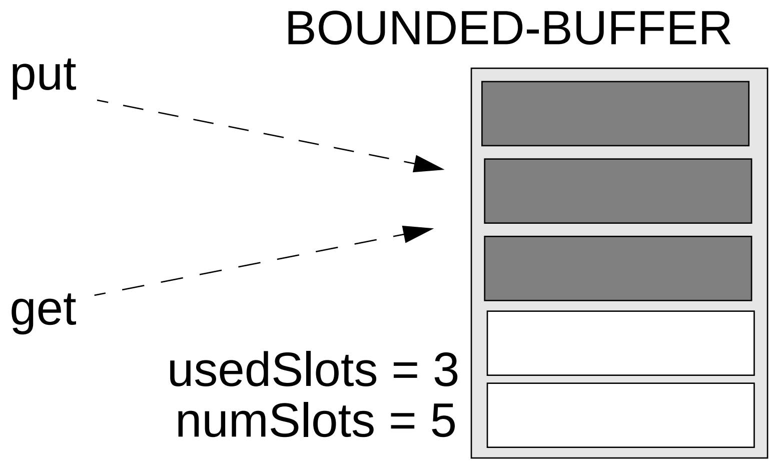


Object Oriented Coordination Languages

The main motivation behind coordination is to allow coordination patterns to be specified separately from the implementation of individual objects.

— cf. Papathomas “ATOM”, pp. 4

Most of the research done in the development of O.O coordination languages have focus in the design of mechanisms for the specification and reuse of per-object synchronization constraints (i.e. per-object coordination).



conditions=
put: if buffer != full
get: if buffer != empty
states =
full = usedSlots != numSlots
empty = usedSlots == 0

Recently, we have a lot of work in multi-object coordination (i.e coordination of group of objects). In SCG we have for example FLO/C: an O.O. coordination language for active objects based on the constraint of methods invocations.

FLO/C

This languages provide abstractions called Connectors that specify and prescribe the inter-object coordination. Connectors react upon messages sends to the objects they control. Their reaction is coded into rules.

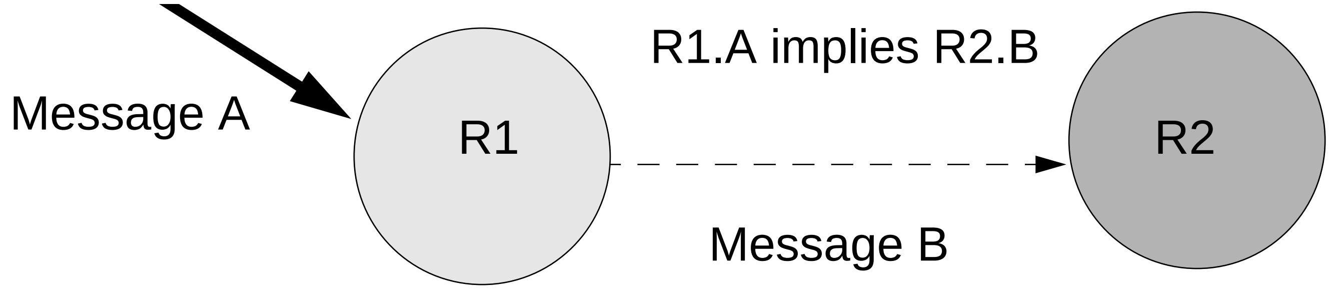
def **connector** <name> (Role [,Role]*)

Rule = Precondition Operator Consequences [; Rule]*

Operators:

Role1.MessageA (arg) Operator Role2.MessageB(arg)

- ❑ impliesLater: Asynchronous Communication
- ❑ implies: Synchronous Communication
- ❑ impliesBefore: Synchronous Communication
- ❑ permittedIf: Conditional Synchronization
- ❑ waitUntil: Conditional blocking Synchronization



Example:

```
class Buffer {
   Vector buff = Null;
   int buffsize = 0;

   public Buffer(int size){
    buff = new Vector(size);
    buffsize = size;}

   public void put(Object x) {
    buff.addElement(x); }

   public Object get() {
    Object x = buff.firstelement();
    buff.removeElement(x); return x;
   }

   public boolean full(){
    return buff.size()==buffsize;}

   public boolean empty() {
    return buff.size()==0;}
}
```

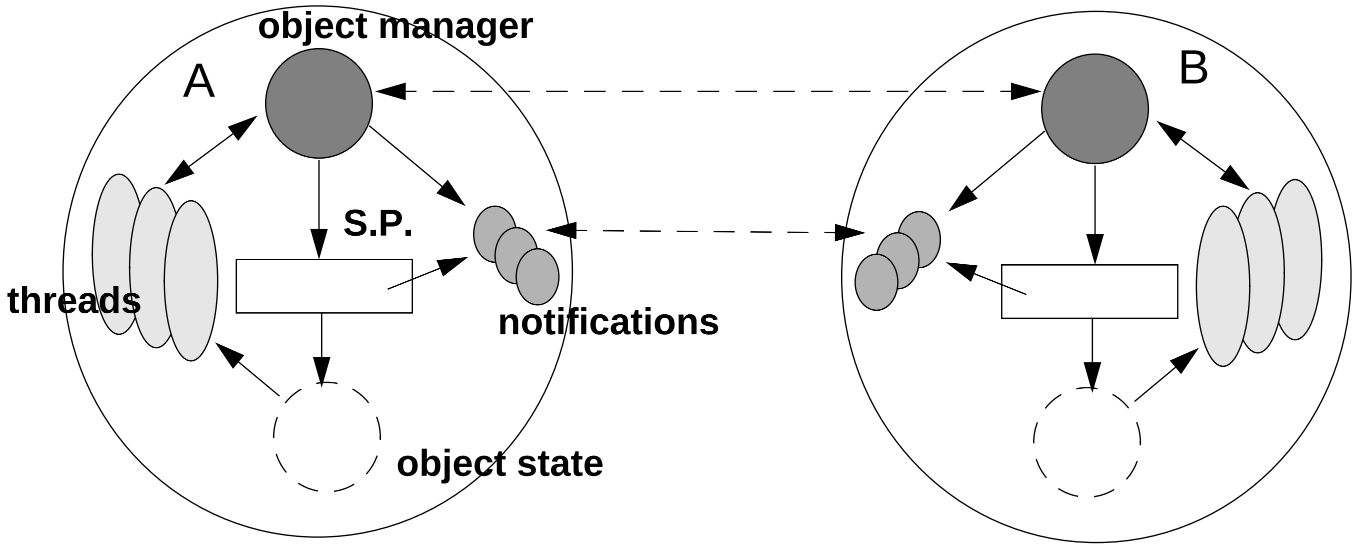
```
def connector ProdCons(Bu,Pr,Co):
   Pr.produce(x) implies Bu.put(x)
   Pr.produce(x) permittedif !Bu.full()
   Pr.consume() implies Bu.get(return x)
   Pr.consume() permittedif !Bu.empty()

class Producer extends Runnable {
   private Buffer buffer = Null;
   public Producer(Buffer buf) {
      buffer = buf;}
   public void produce(Object x) {}
}

class Consumer extends Runnable {
   private Buffer buffer = Null;
   public Consumer(Buffer buf) {
      buffer = buf;}
   public Object consume() {}
}
```

ATOM

A Coordination Language for Active Objects based on State Notifications

Active Object = Objects are active entities that can process the request from other objects, they can delay requests and process them in an order that is most suitable to them (i.e they have control over the synchronization of concurrent request).

- ❑ In ATOM request are processed by threads that execute quasi-concurrently within an object



Example: Per-Object Synchronization

```
from ao import *

class BoundedBuffer(ActiveObjectSupport):

  states  = ['empty','full']

  methods = ['put', 'get']

  conditions = { 'put': (lambda o:not o.atState(('full',)),),
                 'get': (lambda o:not o.atState(('empty',)),)}

  def __init__(self,size):
    self.inbuffer = 0
    self.lim = size
    self.store = []

   def empty(self,state):
    return self.inbuffer == 0
```

Abstract States

Synchronization Constraints

State Empty

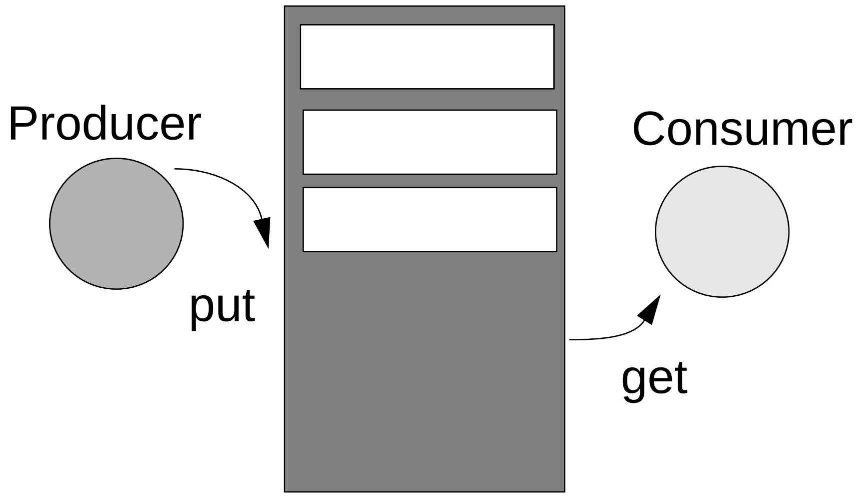
```
    def full(self,state):
      return self.inbuffer == self.lim

    def put(self,data):
      self.store.append(data)
      self.inbuffer = self.inbuffer + 1

    def get(self):
      self.inbuffer = self.inbuffer -1
      d = self.store[0]
      del self.store[0]
      return d
```

State Full

```
Python 1.4 (Jun 4 1997) [GCC 2.7.2]
>>> b = ActiveObject(BoundedBuffer)(10)
>>> b.put(3) x
>>> b.get()
```

Example: Inter-Object Coordination

```
from ao import *

class BoundedBuffer(ActiveObjectSupport):
...
  methods = [‘put’, ‘get’, ‘capacity’]
...
  def capacity(self):
    return self.lim
```



```
Python 1.4 (Jun 4 1997) [GCC 2.7.2]
>>> b = ActiveObject(BoundedBuffer)(10)
>>> p = ActiveObject(Producer)(b)
>>> c = ActiveObject(Consumer)(b)
```

“The producer only produces when the buffer is empty and the consumer only consumes when the buffer is full”

```
class Consumer(ActiveObjectSupport):

  def __init__(self, buff):
    self.buff = buff

  def Activity(self):
    self.noti = self.buff.notifyRequest((‘full’,))
    while (1):
      print “consumer I will wait until full”
      self.suspendUntil(self.noti)
      print “consumer now is full”
      for i in range(self.buff.capacity()):
        x = self.buff.get()
        print “I’m getting from the buffer”,i
      self.noti = self.buff.notifyRequest((‘full’,))
```

Notification Request from the Consumer to the Buffer

Empties the Buffer

```
class Producer(ActiveObjectSupport):

  def __init__(self, buff):
    self.buff = buff

  def Activity(self):
    self.noti = self.buff.notifyRequest((‘empty’,))
    while(1):
      print “producer I will wait until empty”
      self.suspendUntil(self.noti)
      print “producer Is empty”
      for i in range(self.buff.capacity()):
        self.buff.put(i)
        print “I’m putting in the buffer”,i
      self.noti = self.buff.notifyRequest((‘empty’,))
```

Notification Request from the Producer to the Buffer

Fills the Buffer

SCG Coordination Research[1]

Foundations

1. Coordination Patterns and Architectures
   - ☞ Classification of coordination abstractions, patterns and architectural styles
2. Composition Contracts for Concurrent Objects
   - ☞ Composition Contracts

Components

3. Component-Oriented Approach to Coordination:
   - ☞ CoCo: A coordination-component framework

Language Design and Experiments

1. A scripting language for CORBA
2. FLO/C: a coordination language for object-oriented systems.
3. Piccola: a Small Composition Language
4. ProCoordBroker: A Programmable Coordination Broker

Applications

1. Re-engineering: **FAMOOS**[2]

1. ESPRIT Working Group 24512 - “Coordina: Coordination Models and Languages”
2. ESPRIT Project 21975 — “Framework-based Approach for Mastering Object-Oriented Software Evolution”

8. Coordination Components in Java

Overview

- ❑ The context
  - ☞ Coordination for Open Distributed Systems
  - ☞ Components
- ❑ Communication components: network communication
- ❑ Synchronization components: shared resource
- ❑ Composing abstractions: distributed coordination
  - ☞ distributed shared resource
  - ☞ distributed transactions

Sources

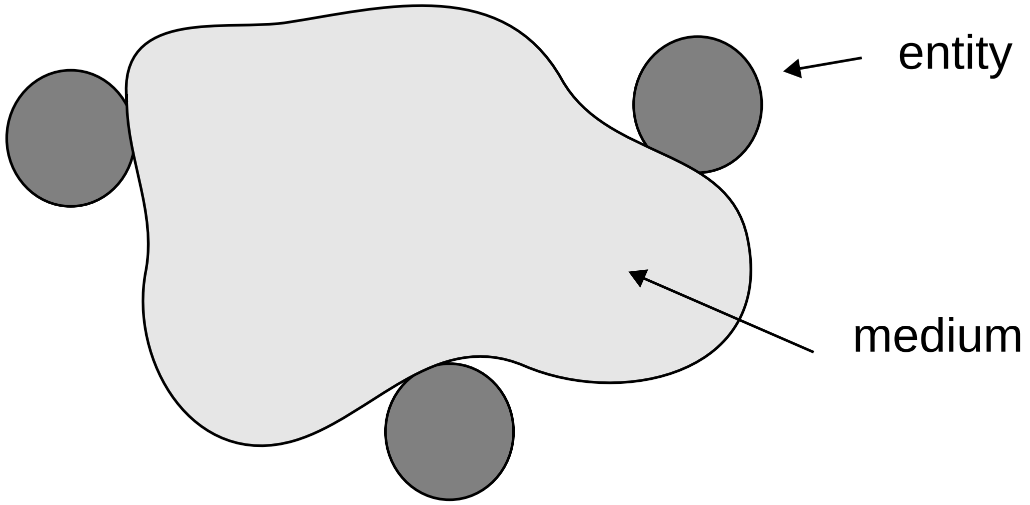
- ❑ Gamma et al., *Design Patterns*, Addison-Wesley, 1994.
- ❑ D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996.

Coordination for Open Distributed Systems

Coordination *is the act of managing the interaction between activities of entities*

An **Open Distributed System** *is a collection of loosely coupled entities in a distributed environment, working together to achieve a goal. The system is extendable and heterogeneous.*

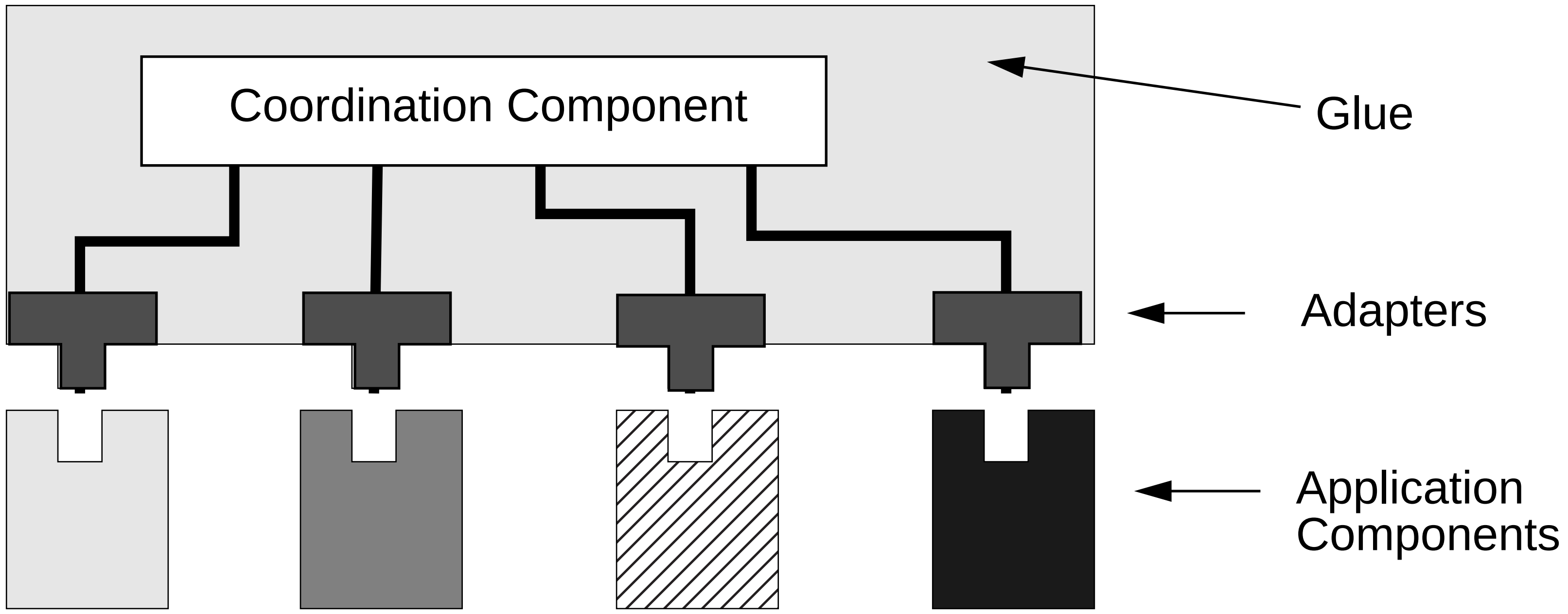
Programs = Coordination + Computation



Approach: Coordination Components

A **Component** *is a generic blackbox abstraction which is reconfigurable and composable by plugs*

Glue *is the element of programming concerned with “putting things together”*

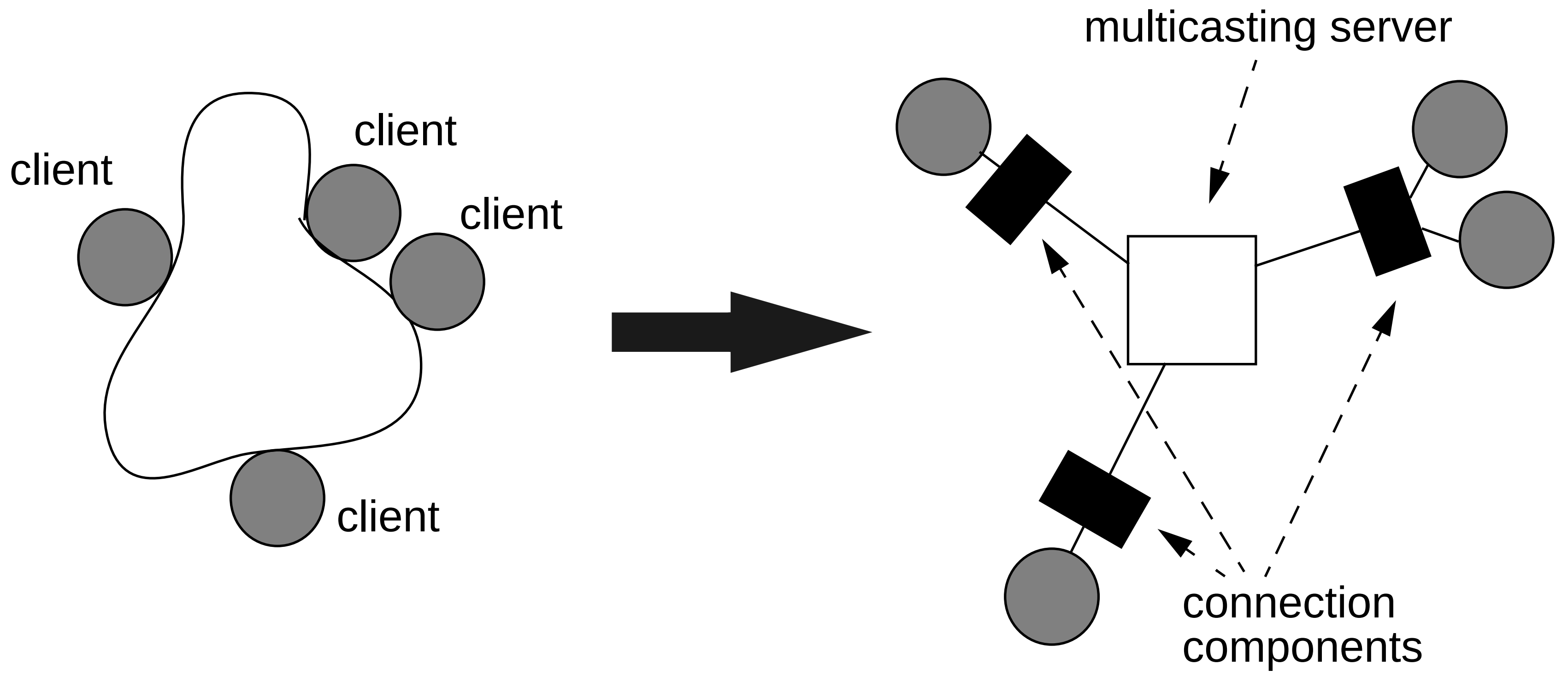


✔ *Goal: an open, flexible and reusable coordination layer*

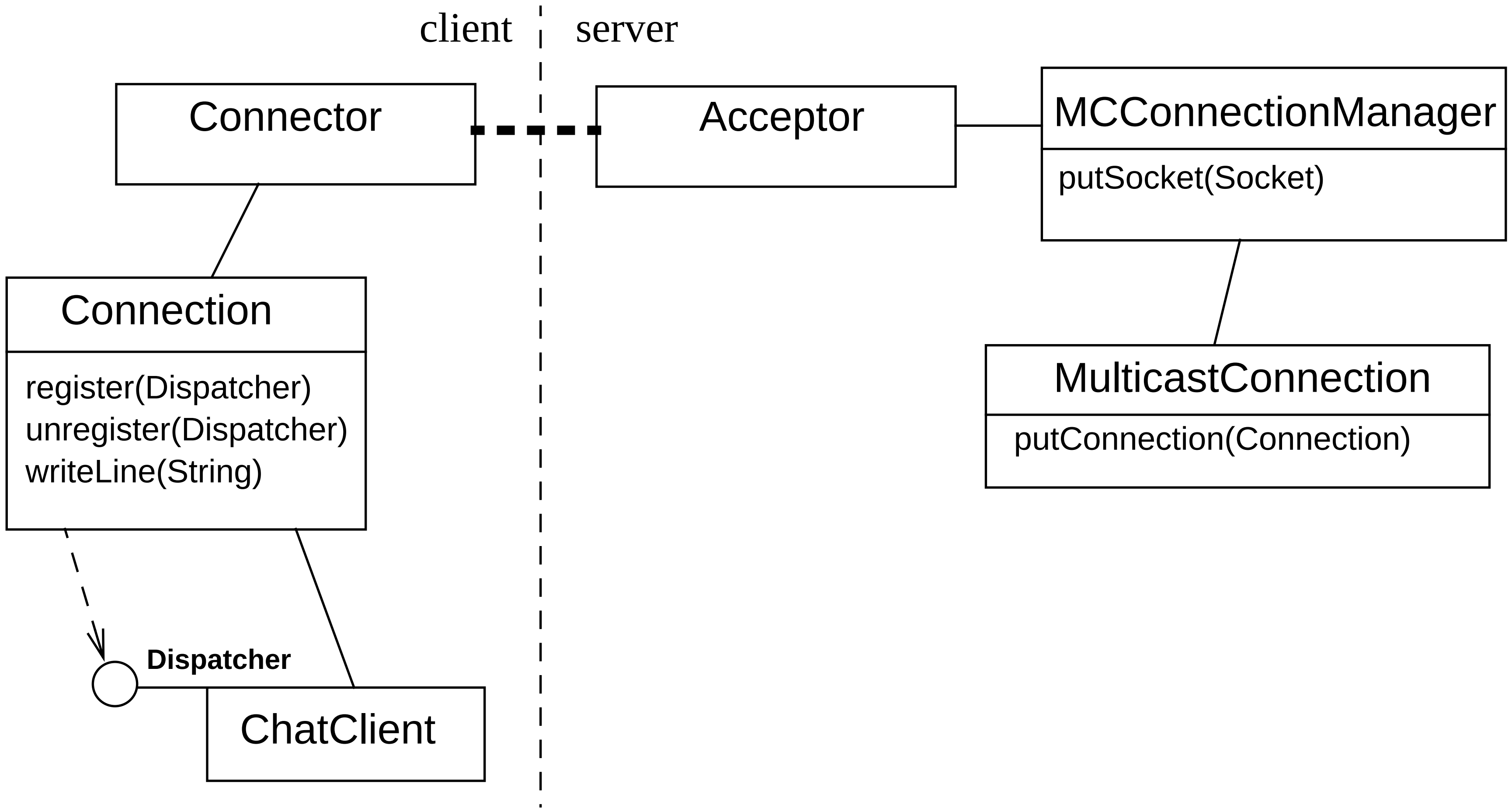
Network communication

Coordination Problem: how to distribute messages over a network to a set of interested clients?

Example: a chat environment that multicasts the lines of the participants to all other participants

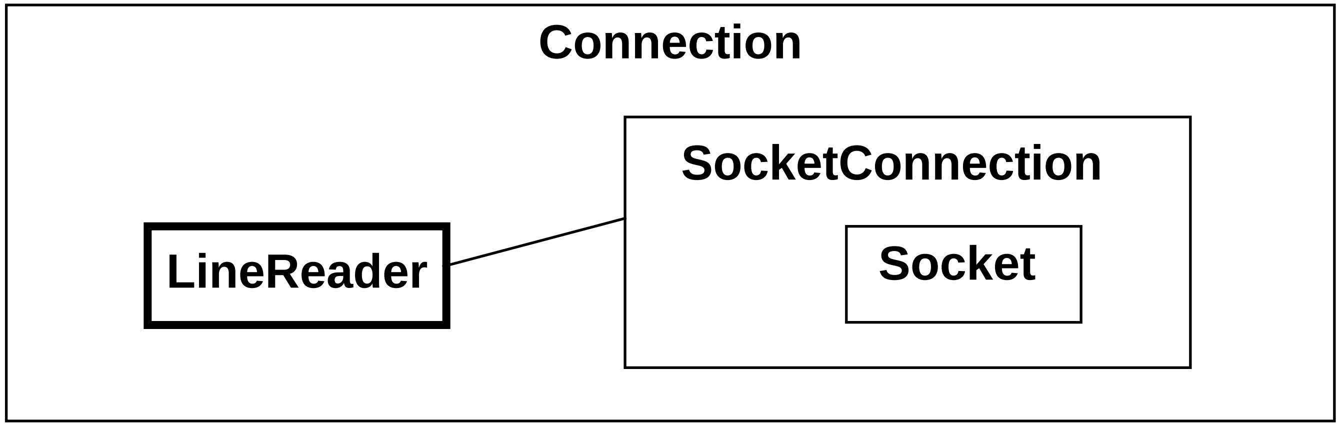


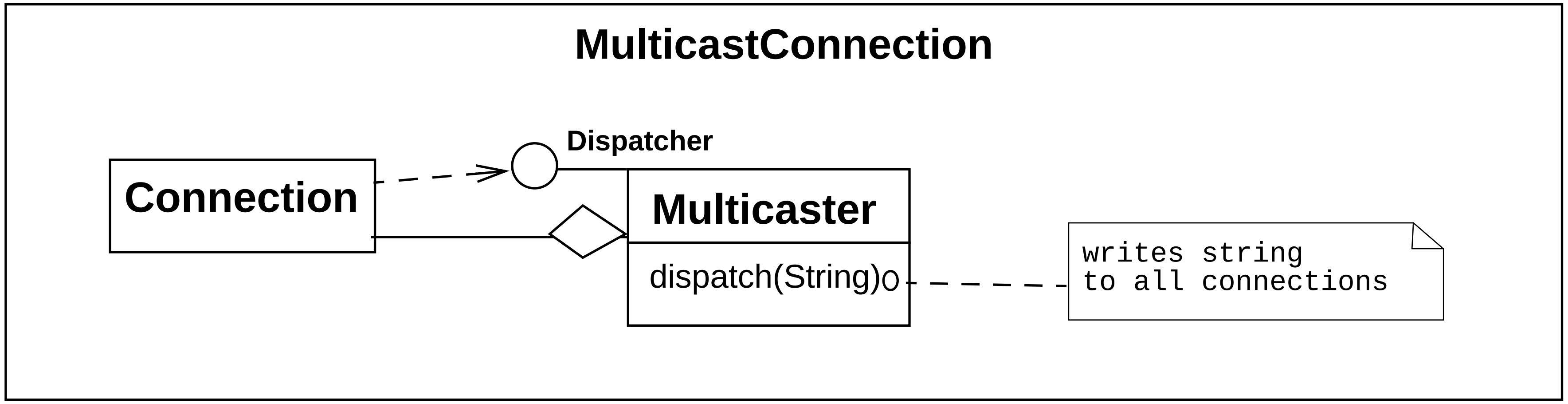
A design for the multicast solution



- ✔ *Connector and Acceptor take care of setting up the connection*
- ✔ *MulticastConnection multicasts to all connected clients*
- ✔ *Dispatcher interface ensures that there is a* dispatch(String line) *method*

The Connection components



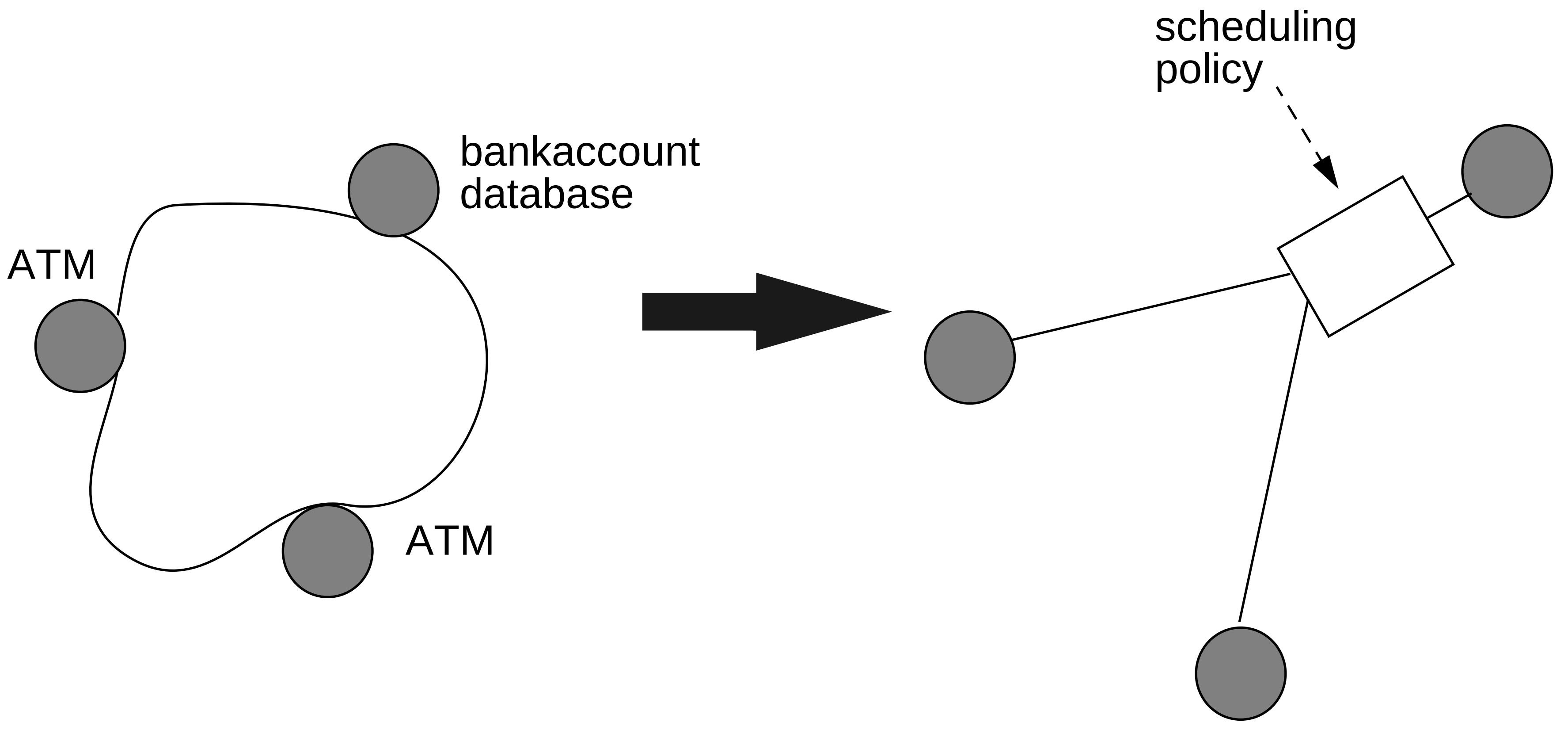


The LineReader in Java

```
public class LineReader implements Runnable {
...
   public void run() {
      for(;;) {
       line = conn.readLine(); // waits for line

       class LineHandler implements Runnable {
         public LineHandler(Dispatcher target, String line) {
            (new Thread(this)).start();
         }
         public void run() { target.dispatch(line); }
      }

       new LineHandler(target, line);
    }
   }
}
```

Synchronization: Shared Resource

Coordination Problem: how to deal with concurrent requests that want to access a resource (while keeping the resource consistent)?



Requirements for scheduling policy

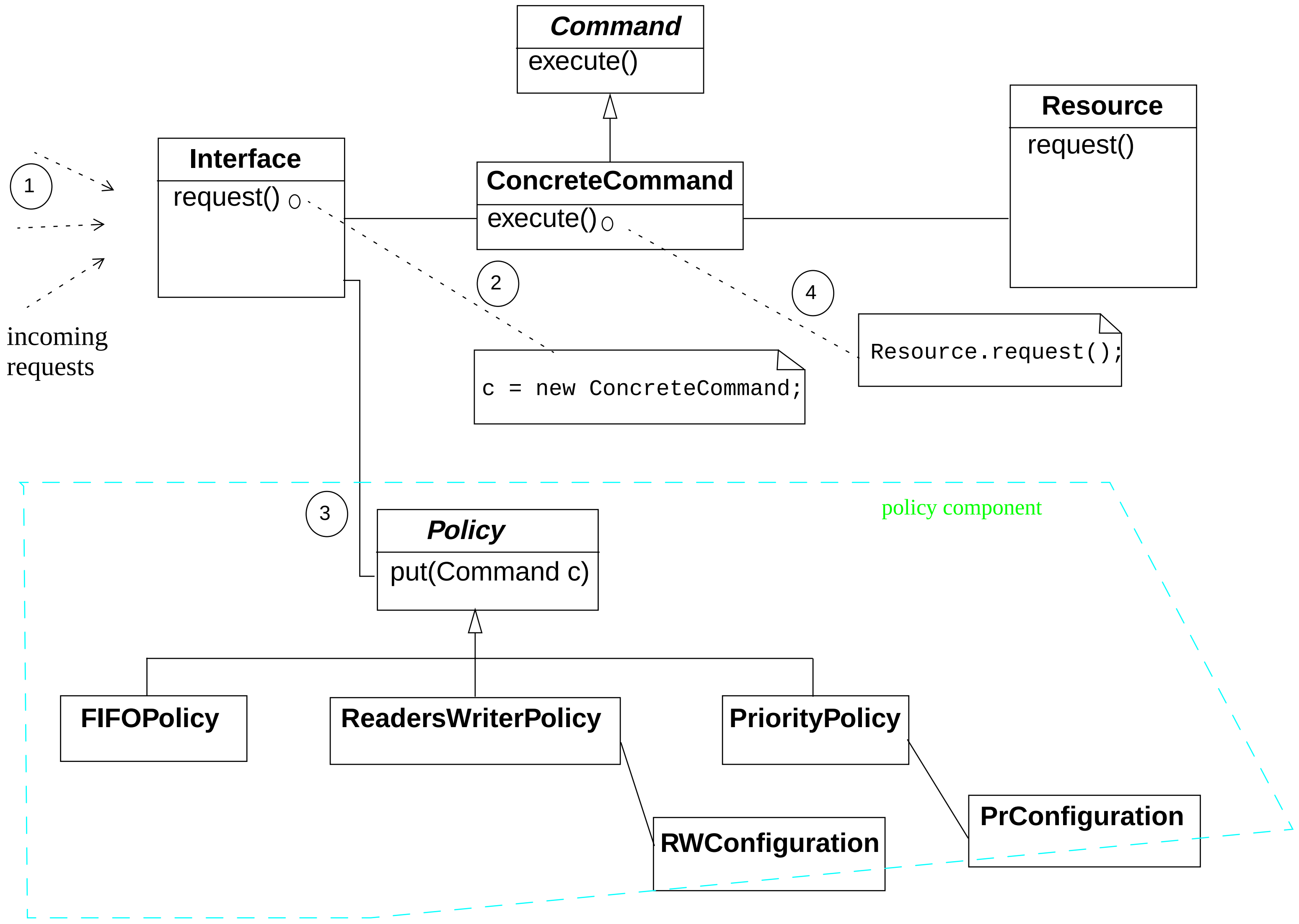
- ❑ dispatch concurrent requests
- ❑ resource stays consistent
- ❑ bank and database independent of policy
- ❑ policy independent of bank and database

- ✔ *why do we want this independence?*
- ✔ *what control policies are possible?*
- ✔ *what are possible problems concerning the independence?*

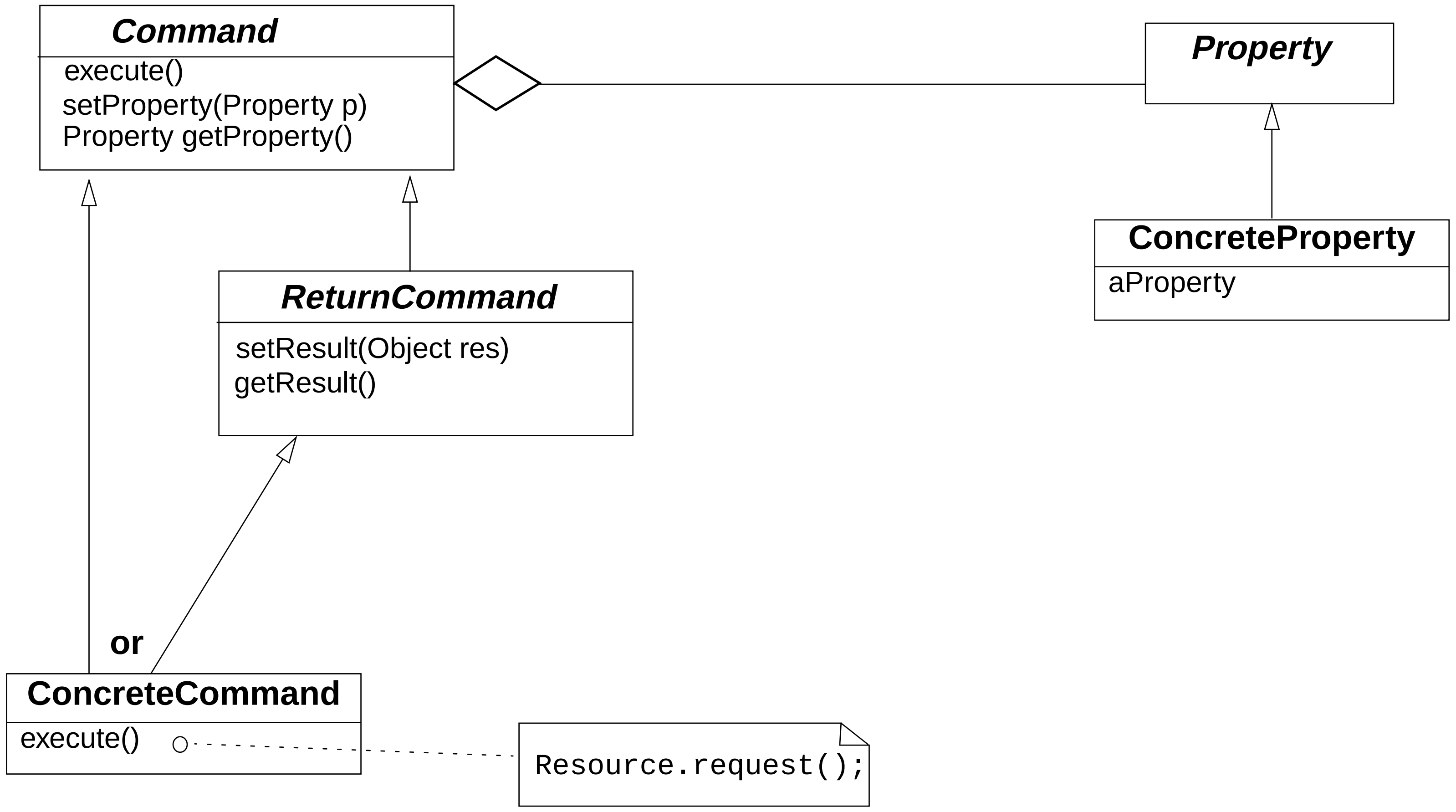
The solution consists of a set of design and concurrency patterns:

- ➪ explicit commands (Command pattern)
- ➪ explicit policy (Policy pattern (aka Strategy pattern))
- ➪ explicit properties
- ➪ early replies
- ➪ different synchronization policies
- ➪ configuration objects

Scheduling Policy Design

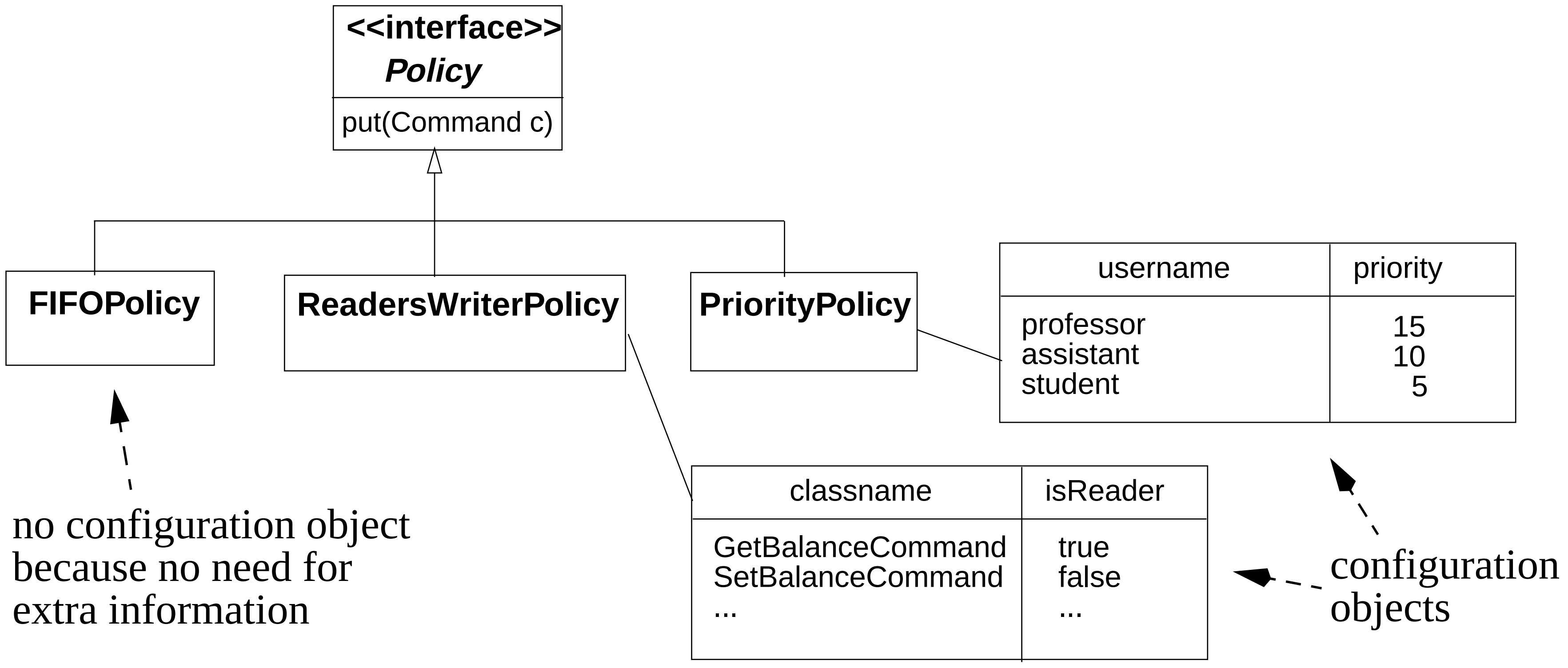


Explicit Commands



- ✔ *uses the Command pattern (see Gamma et al)*
- ✔ *the ReturnCommand uses the early reply mechanism (see slide 72 a.f.)*

Explicit Policies



username	priority
professor	15
assistant	10
student	5

classname	isReader
GetBalanceCommand	true
SetBalanceCommand	false
...	...

✔ *the policies are not all completely independent: some need application specific information !!*

The Configuration Object

classname	isReader
GetBalanceCommand	true
SetBalanceCommand	false
...	...

```
Hashtable table = new Hashtable(); // available in java.util
public void add(String classname, boolean isReader) {
 table.put(classname,new Boolean(isReader));
}
public void remove(String classname) {
 table.remove(classname);
}
public boolean isReader(Command c) {
   return ((Boolean)table.get(c.getClass().getName())).booleanValue();
}
```

The adapted readers/writer policy

The policy of slide 94 is slightly adapted. Instead of:

```
public void read() {
   beforeRead(); read_(); afterRead();
}
public void write() {
   beforeWrite(); write_(); afterWrite();
}
```

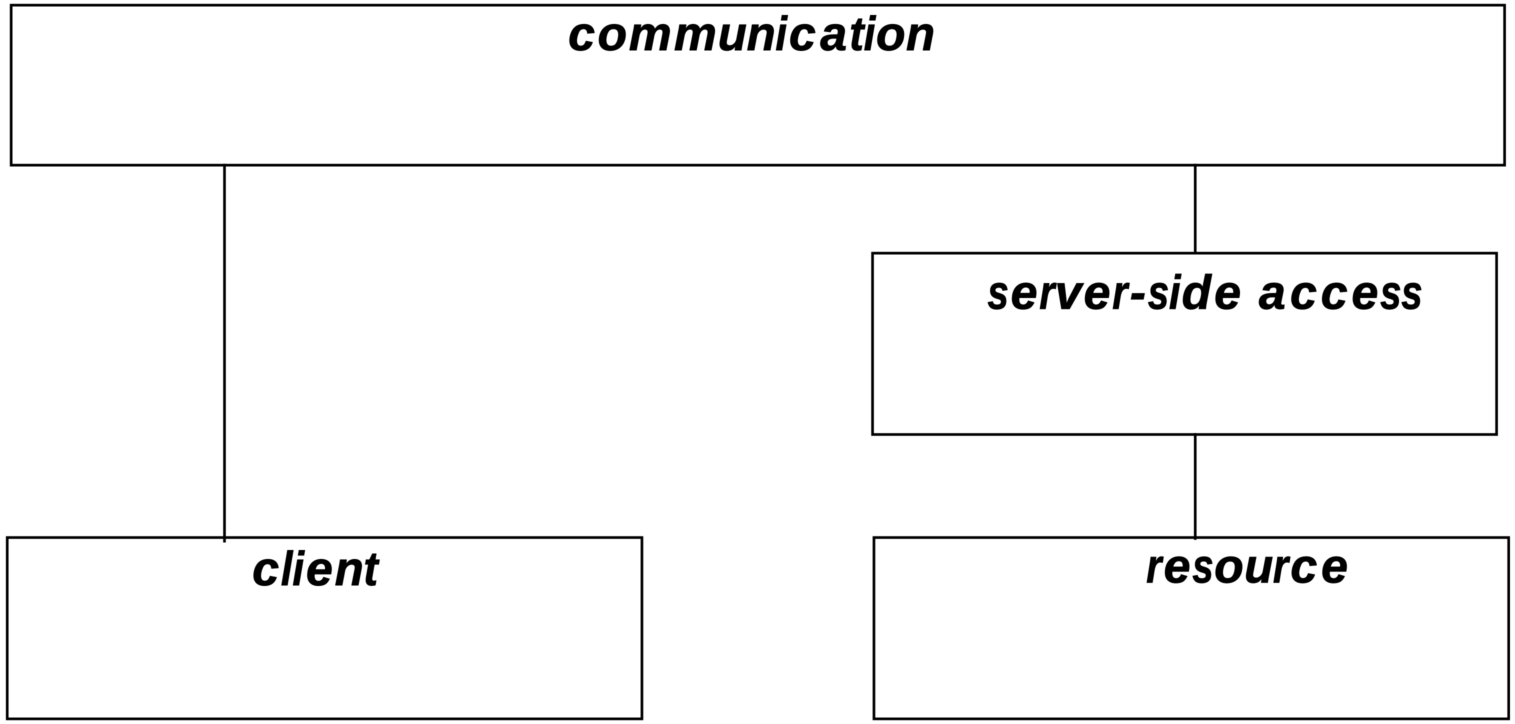
we now have:

```
public void dispatchReadCommand(Command c)  {
   beforeRead(); c.execute(); afterRead();
}
public void dispatchWriteCommand(Command c) {
   beforeWrite(); c.execute(); afterWrite();
}
```

Which enables the policy to give every command its own thread and execute it according to the RWVT policy implementation:

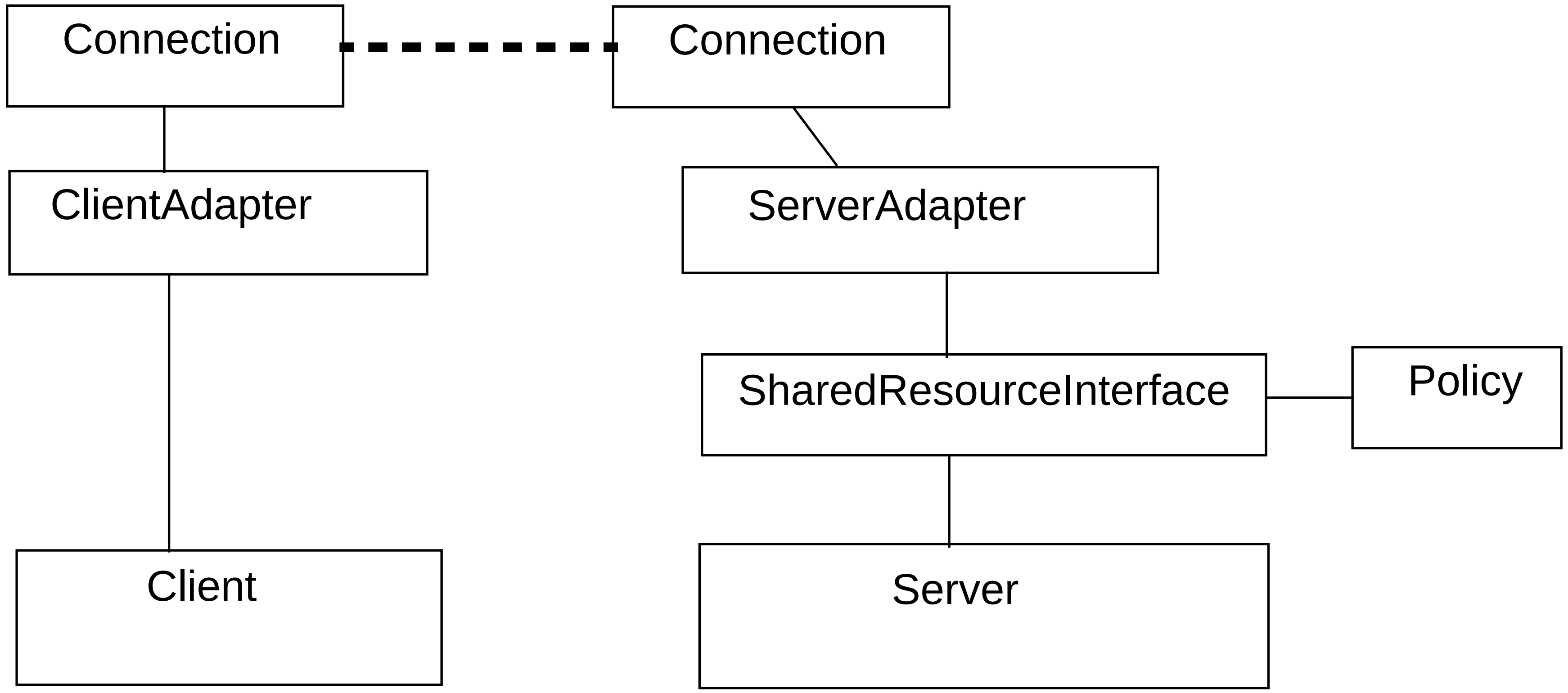
```
public class ReadersWriterPolicy implements Policy {
   private RWConf conf; // configuration object
   private RWVT rwvt = new RWVT();  // actual policyholder
   public ReadersWriterPolicy(RWConf conf) { this.conf = conf;}

   public void put(Command c) {
      class RWCommandHandler implements Runnable {
      ....
         public RWCommandHandler(Command cmd,  boolean isReader) {
            this.cmd = cmd; this.isReader = isReader;
            (new Thread(this)).start();
         }
         public void run() {
            if (isReader) { rwvt.dispatchReadCommand(cmd); }
            else { rwvt.dispatchWriteCommand(cmd);}
         }
      }
      new RWCommandHandler(c,conf.isReader(c)); }}
```

Composing abstractions: distributed access



✔ *a simple layered architecture*

A design for a distributed shared resource



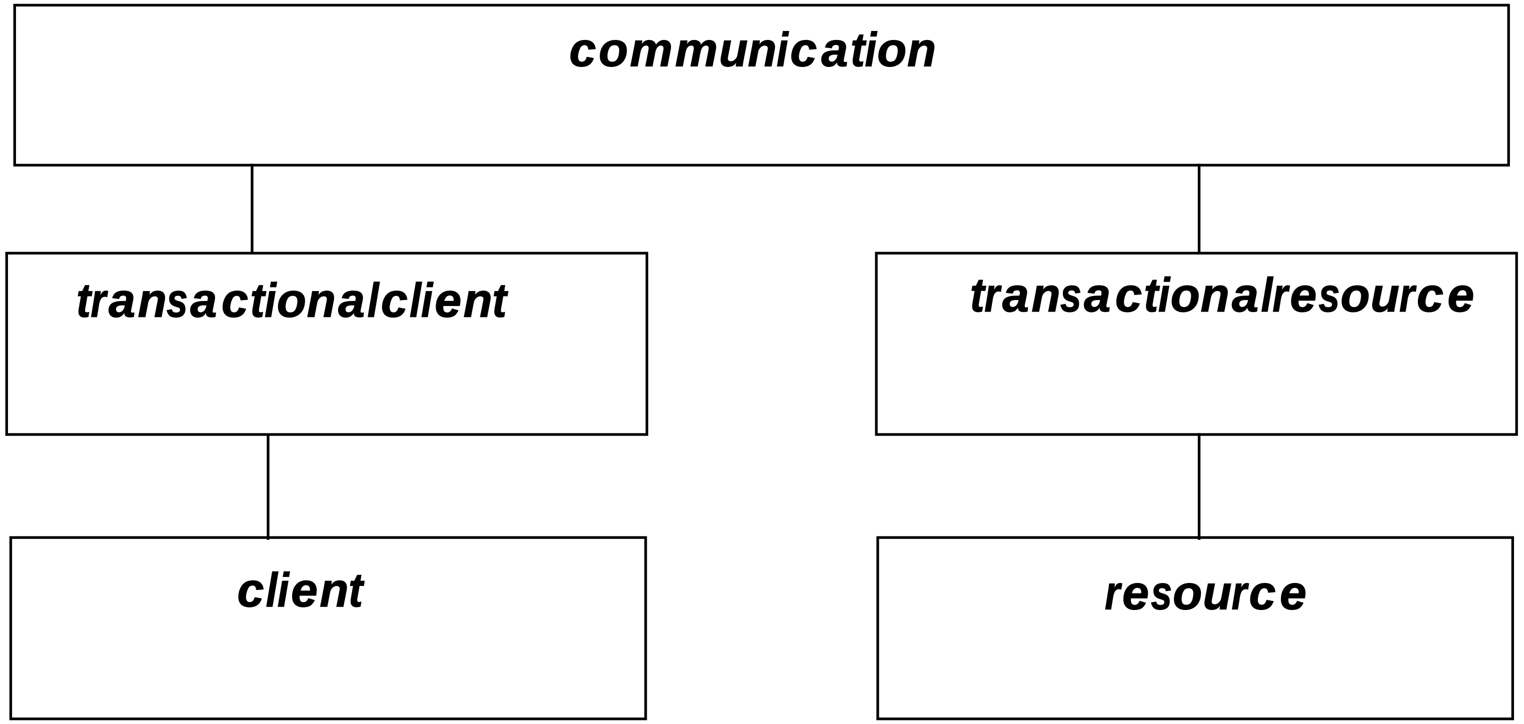
✔ *the adapter converts, for instance, the synchronous calls of the lockpolicy to the asynchronous calls of the communication layer. The adapter could be converting to CORBA or RMI or whatever communication mechanism as well*

An Adapter in Java

```
public class ServerAdapter implements Dispatcher {
   Connection conn;
   ResourceInterface bank;
...

   public void dispatch(String line) {
      java.util.StringTokenizer strtok = new java.util.StringTokenizer(line,"&");
      if (strtok.nextToken() == “getBalance”) {
         int value = bank.getBalance();
         conn.writeLine(“&getBalance&”+value);
      }
      else if ....
   }
}
```

- ✔ *this adapter converts from asynchronous line sends to synchronous method calls*
- ✔ *line can have more information (for instance, parameters like account number) that will have to be parsed as well*

Distributed transactions



✔ *In a (distributed) transaction the client has first to grep a set of resources before it can do actions on them. If something goes wrong during the transaction it is rolled-back: the initial state of the participants is restored.*

Transactions in a nutshell

The client view:

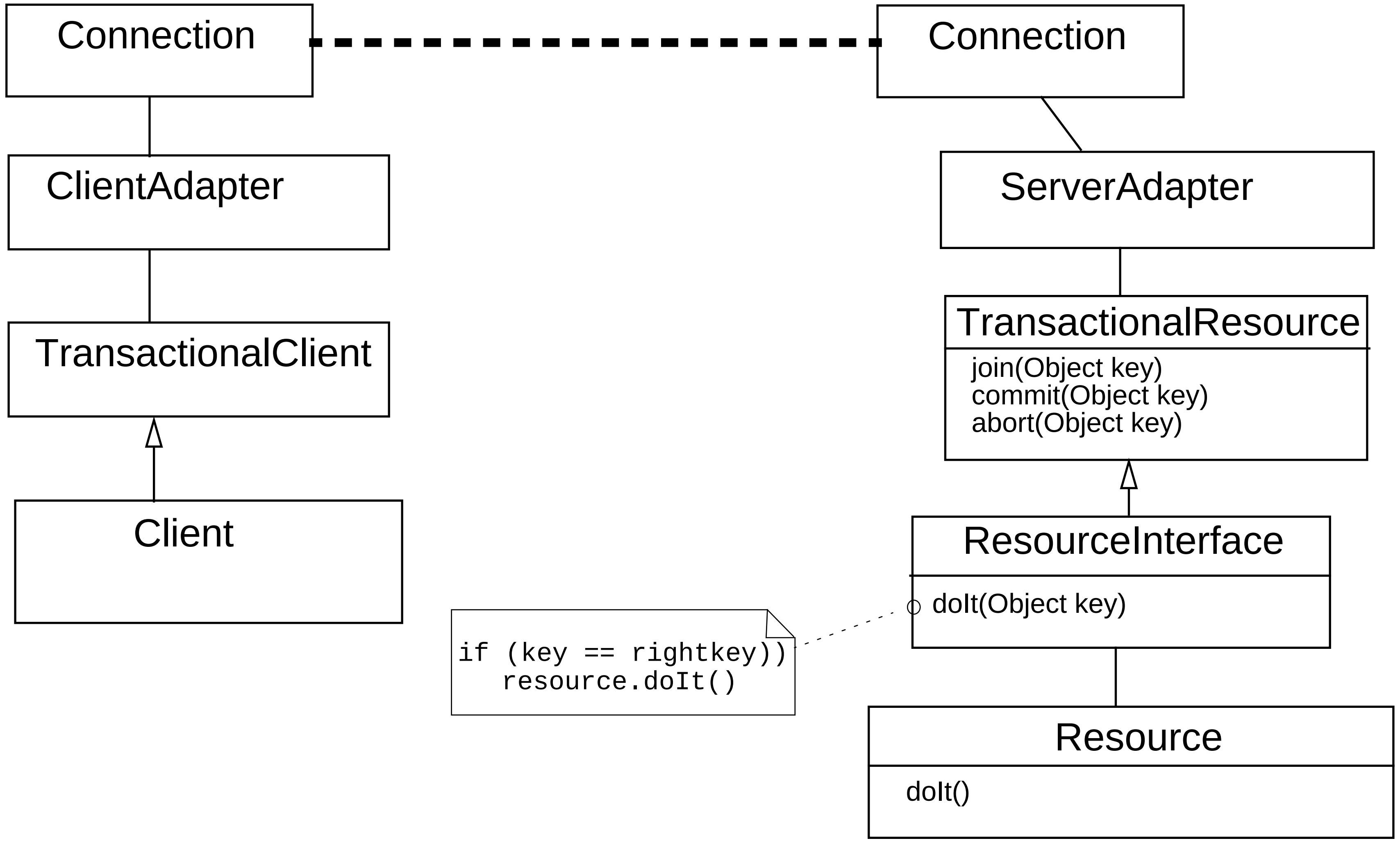
idea

```
atomic {
   resource1.doIt();
   resource2.doIt();
}
```

implementation in Java

```
try {
   Object key = new Object();
   resource1.join(key);
   resource2.join(key);

   resource1.doIt(key);
   resource2.doIt(key);

   resource1.commit(key);
   resource2.commit(key);
}
catch(TransactionException e){
   resource1.abort(key);
   resource2.abort(key);
}
```

a design for distributed transactions



```
if (key == rightkey))
   resource.doIt()
```

✔ *operations to organize recover or resource state are not shown*

9. Object-Based Concurrency

Overview

- ❑ What is an OBCL?
- ❑ Dimensions of OO Languages
- ❑ Expression of Concurrency
  - ☞ Objects and Processes
  - ☞ Granularity of Concurrency
  - ☞ Creating Processes
- ❑ Communication and Synchronization
  - ☞ Intra-Object and Inter-Object Synchronization
- ❑ Evaluating OBCLs
- ❑ Research Topics

What is an OBCL?

An Object-Based Concurrent Language supports:

- ❑ Encapsulation
  - ☞ objects encapsulate data and operations
- ❑ Concurrency
  - ☞ multiple processes may be concurrently active
  - ☞ need to: specify, create and synchronize processes

Why do we need OBCLs?

- ❑ Inherent application (real-world) concurrency
- ❑ Distributed applications
- ❑ Application integration and interoperability
- ❑ Parallel applications

Overview of OBCLs

- Traditional OBLs:
  - Smalltalk-80, C++, Objective C, Ada
  - libraries
- Extended OBLs:
  - CLU: Argus
  - Smalltalk-80: ConcurrentSmalltalk, Actalk, PO
  - C++: ACT++, Arjuna, Avalon, Karos
  - Eiffel//
- Concurrent OBLs:
  - Actors, ABCL, POOL, Guide, Hybrid, Java

Requirements for OBCLs

- ❑ Object autonomy:
  - ☞ protection from concurrent requests
- ❑ Internal concurrency:
  - ☞ should be transparent to clients
- ❑ Local delay transparency:
  - ☞ handling of local delays should be transparent to the client
- ❑ Remote delay transparency:
  - ☞ handling of remote delays should be transparent to the service provider
- ❑ Composable synchronization policies:
  - ☞ subclasses should share synchronization code with superclasses

REF: *Papathomas, PhD thesis, 1992.*

Expression of Concurrency

- ❑ Objects and Processes:
  - ☞ How are processes and objects related?
- ❑ Granularity of Concurrency:
  - ☞ How many processes can be associated with an object?
- ❑ Process Creation:
  - ☞ How are processes created?

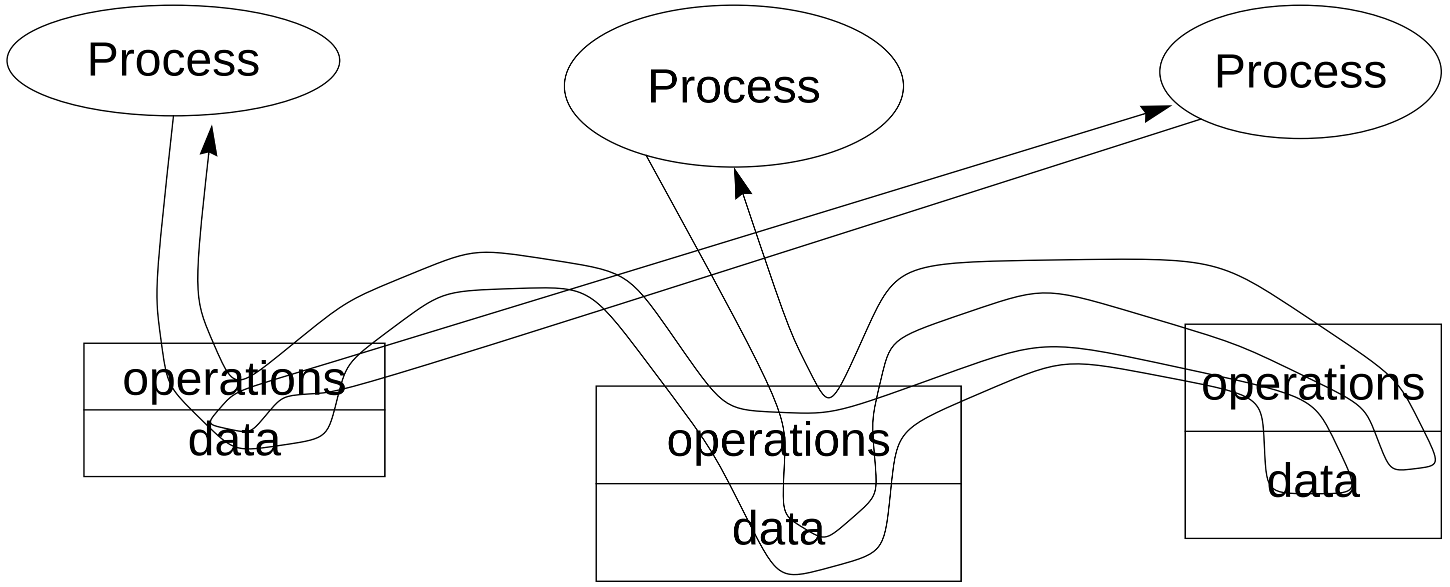
Objects and Processes

How are processes related to objects?

Three Classes of OBCL:

❑ *Passive Objects:*	objects & concurrency independent (Smalltalk-80, C++, Objective-C, Emerald, Java)
❑ *Active/Passive:*	passive + “concurrent” objects (PAL)
❑ *Active Objects:*	objects and processes are unified (ABCL/1, Hybrid, POOL ...)

Passive Object Models



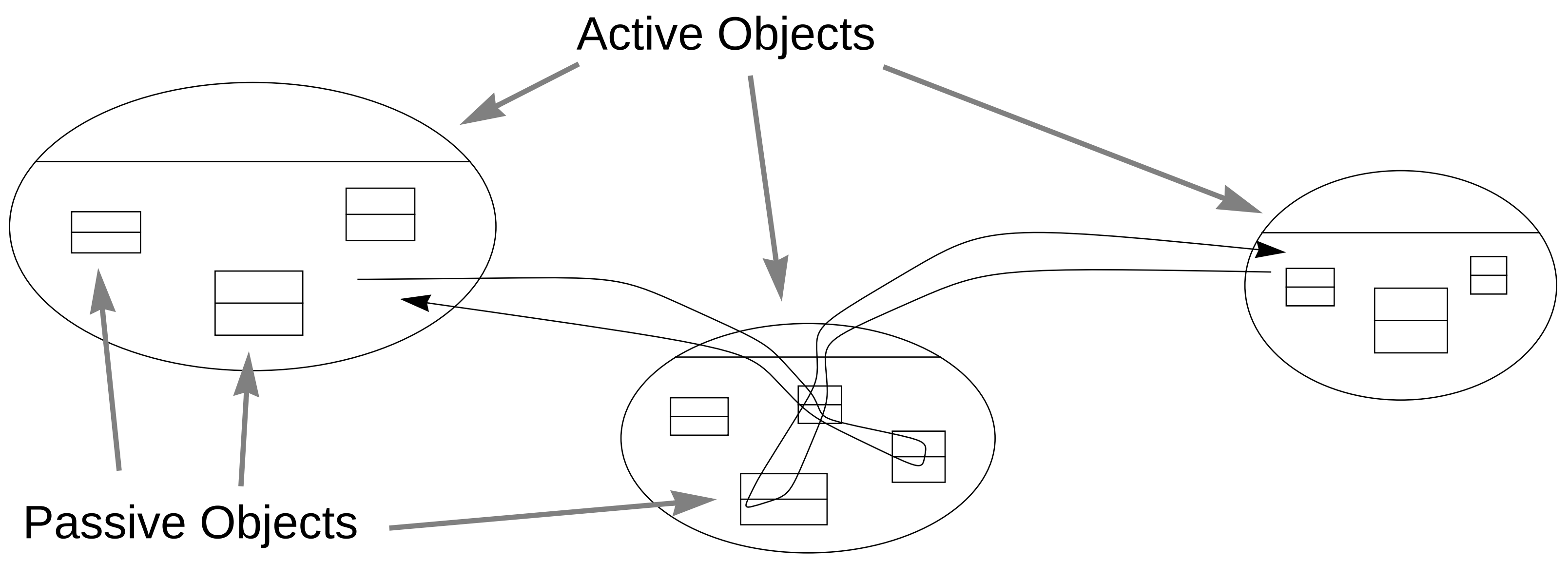
Concurrent processes access passive objects.

Processes synchronize according to a shared memory model:

- ☞ objects must be designed to be shared, or
- ☞ processes must explicitly synchronize via locks, etc.

Smalltalk-80, C++, Objective-C, Emerald, Java

Active/Passive Models



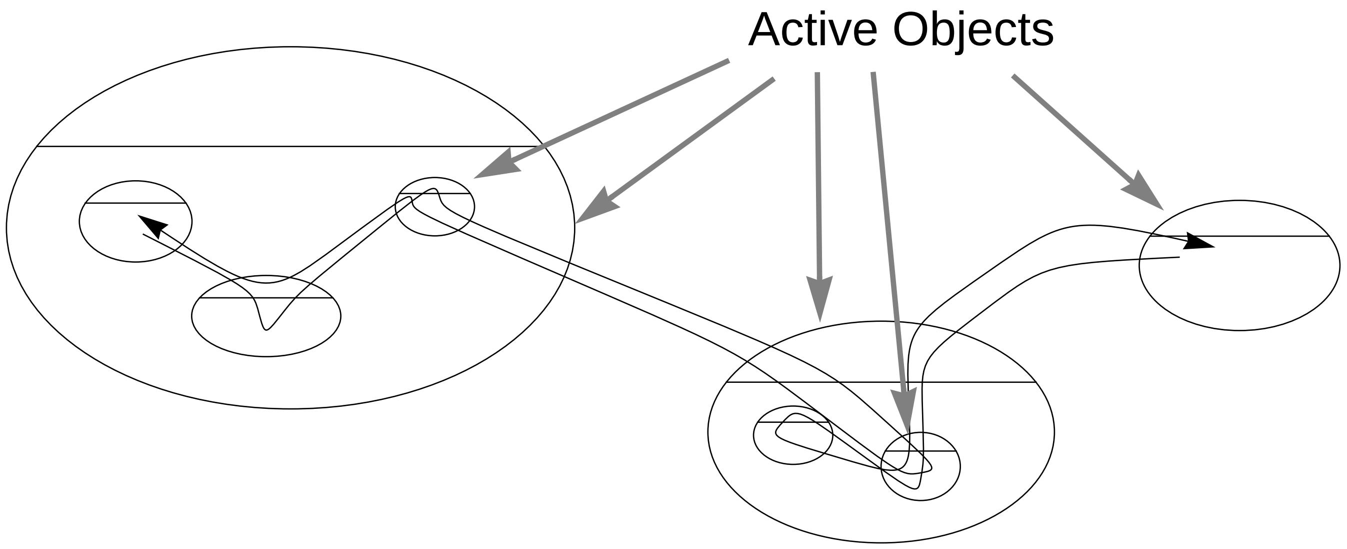
Active Objects are identified with processes

Passive objects are protected by the active objects containing them

- ☞ lightweight/heavyweight distinction
- ☞ two class hierarchies are incompatible

PAL

Active Object Models



Objects and processes are integrated:

- ☞ each operation invocation is a potentially concurrent thread
- ☞ an object with a running operation is *active*
- ☞ every object is autonomous and synchronizes its own threads

ABCL, Hybrid, POOL, ...

Granularity of Concurrency

Approaches to Concurrency:

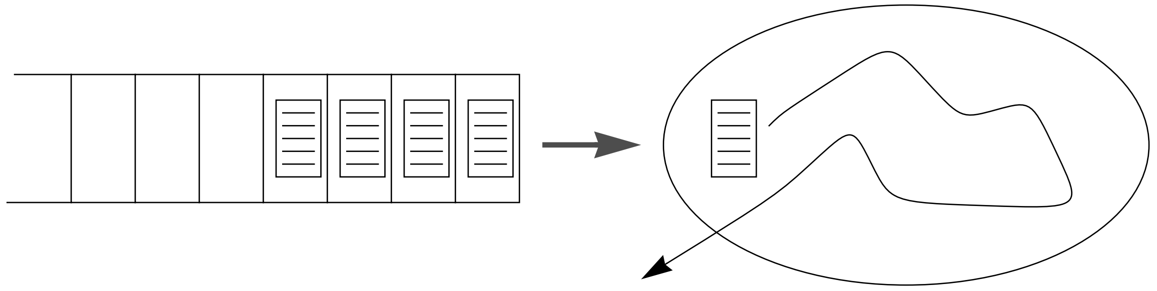
Inter-Object Concurrency:

❑ Sequential Objects	*Ada, POOL*

Intra-Object Concurrency:

❑ Quasi-Concurrent Objects	*ABCL, Hybrid*
❑ Concurrent Objects:	
☞ Client-Driven: Passive Objects	*Smalltalk, Java*
☞ Server-Driven: Active Objects	*Sina, PO, Eiffel//*

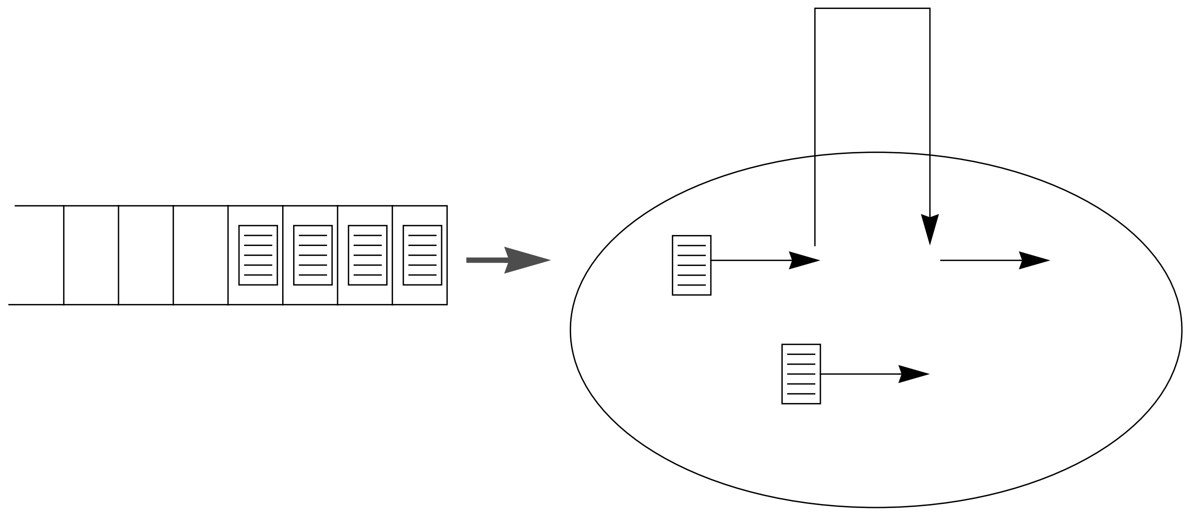
Sequential Objects



In a sequential object model, requests are serialised in a wait-queue

- ☞ each operation runs to completion before the next request is handled
- ☞ concurrency is introduced by having more objects

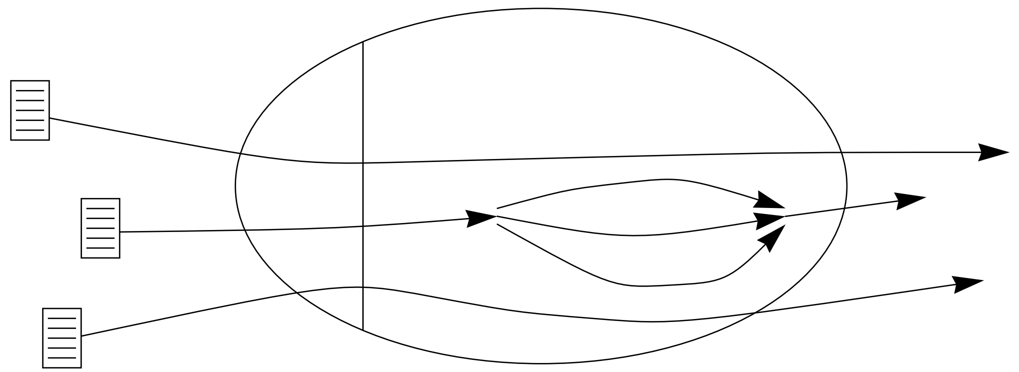
Quasi-Concurrent Objects



Quasi-concurrent objects may switch attention between multiple requests:

- ☞ In *Hybrid*, a *delegated call* to another object allows the serving object to switch to another request
- ☞ In *ABCL*, an *express message* may interrupt the thread servicing an ordinary invocation

Concurrent Objects



Concurrent Objects may serve multiple requests concurrently:

- ❑ Passive Objects require explicit synchronization of threads
- ❑ Active Objects control when to accept new requests
  - ☞ may create additional internal threads to service a single request

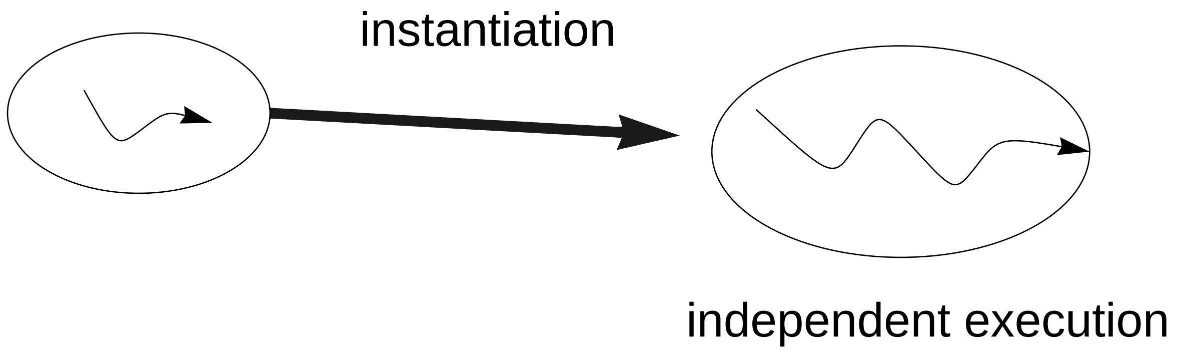
Passive: Smalltalk-80, Java, ...

Active: Sina, PO, Eiffel//, ...

Process Creation

- Asynchronous Objects
  - Explicit bodies
  - Implicit bodies
- Asynchronous Messages
  - one-way message-passing
  - futures

Asynchronous Objects

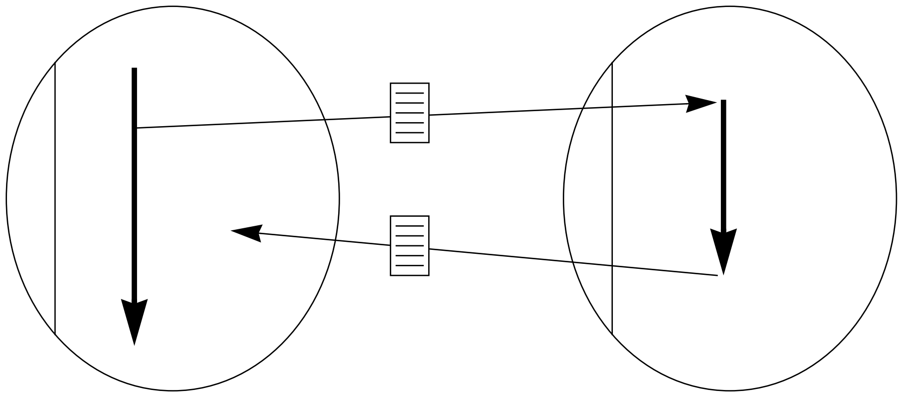


The “body” of an active object may be:

- ☞ Implicit and inaccessible — standard scheduler
- ☞ Explicit and customizable — initialization, scheduling, synchronization ...

Implicit: Actalk, Act++, Actors
Explicit: Ada, Eiffel//, Pool

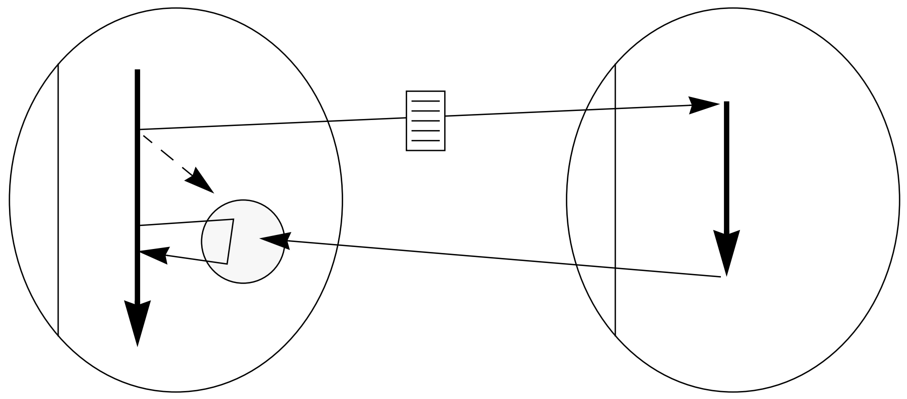
Asynchronous Invocation



Clients do not wait for the reply to continue executing

- ❑ one-way message-passing:
  - ☞ reply (if any) sent by *another* invocation
- ❑ futures:
  - ☞ reply sent to a *future object*

Futures



The reply to an asynchronous request is sent to a *future object*..

- ☞ The client obtains the result when needed.
- ☞ Clients block only if the result is not yet available when needed

Futures may be created either *explicitly* by clients or *implicitly* for all requests.

Explicit: ACT++, ABCL, PO, ConcurrentSmalltalk

Implicit: Eiffel//, Karos, Meld

Communication and Synchronization

- ❑ Intra-Object Synchronization:
  - ☞ Remote Delays: asynchronous invocations
  - ☞ Local Delays: condition synchronization
- ❑ Inter-Object Synchronization:
  - ☞ Transactions

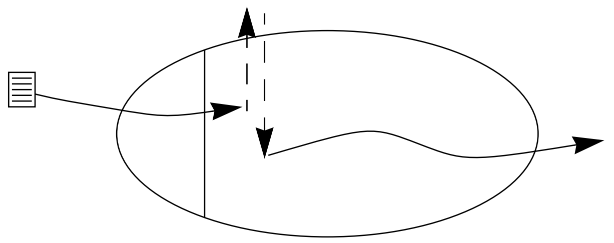
Local Delays

An object may need to delay selected requests to avoid local inconsistency.

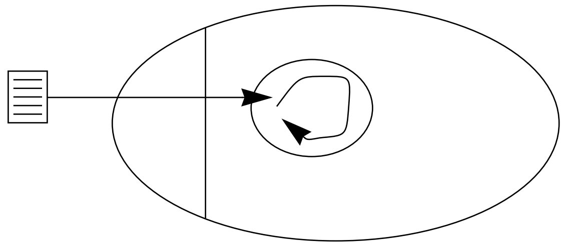
❑ Unconditional acceptance	*Emerald, Smalltalk-80, Java*
❑ Conditional acceptance	
☞ Centralized acceptance	
➪ Explicit acceptance	*Ada, POOL, ABCL*
➪ Reflective computation	*Actalk, ABCL/R*
☞ Distributed activation conditions	
➪ Representation specific	*Guide, Hybrid, SINA*
➪ Abstract	*Procol, ACT++, Rosette*

Local Delays

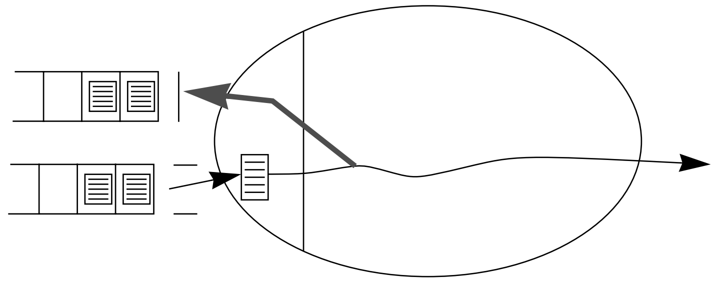
Unconditional acceptance



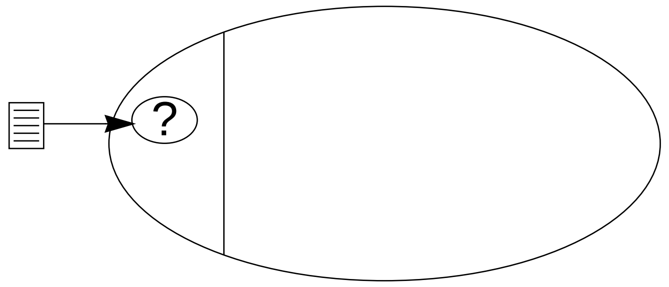
Explicit acceptance



Representation specific delays



Abstract synchronization conditions



Transactions

- ❑ Concurrency atomicity:
  - ☞ intermediate effects on shared objects are invisible to other transactions *(serialisability or isolation)*
- ❑ Failure atomicity:
  - ☞ transactions either complete successfully, or are aborted with no visible effect on shared objects *(the “all-or-nothing” property)*

Transactions may be associated with *transaction blocks* (explicit start and end), or may be realized as *atomic invocations* (implicit with operation start and end).

Classifying OBCLs

- ❑ Object Models
  - ☞ Active or Passive Objects?
- ❑ Granularity of Concurrency
  - ☞ Sequential, Quasi-Concurrent or Concurrent?
- ❑ Process Creation
  - ☞ Asynchronous Objects or Asynchronous Invocations?
- ❑ Local Delays
  - ☞ Conditional or Unconditional Acceptance?
  - ☞ Centralized or Distributed Activation Conditions?
  - ☞ Explicit or Reflective / Abstract or Representation-specific?

Evaluation

- ❑ Object autonomy:
  - ☞ active objects
- ❑ Internal concurrency:
  - ☞ server-driven
- ❑ Local delay transparency:
  - ☞ various approaches ...
- ❑ Remote delay transparency:
  - ☞ futures or internal threads
- ❑ Composable synchronization policies:
  - ☞ composable abstract synchronization policies ...

Summary

You Should Know The Answers To These Questions:

- ❑ What is the difference between active and passive objects?
- ❑ What is the difference between client- and server-driven concurrency?
- ❑ What different ways are there to introduce concurrency in applications?
- ❑ What are local and remote delays?
- ❑ What are the usual ways to implement local delays?
- ❑ How can an object avoid remote delays?

Can You Answer The Following Questions?

- ✎ *What kinds of problems cannot be easily solved with purely sequential objects?*
- ✎ *When is the active/passive model useful when programming in Java?*
- ✎ *How could you implement an active object in Java? Why would you do so?*
- ✎ *How would you implement futures in Java?*
- ✎ *Suppose you want to extend a class that makes use of* `synchronize`*,* `wait()` *and* `notify()` *— what would you have to be careful about in your subclass extensions?*

10. Petri Nets

Overview

- ❑ Definition:
  - ☞ places, transitions, inputs, outputs
  - ☞ firing enabled transitions
- ❑ Modelling:
  - ☞ concurrency and synchronization
- ❑ Properties of nets:
  - ☞ liveness, boundedness
- ❑ Implementing Petri net models:
  - ☞ centralized and decentralized schemes

Sources

- ❑ J. L. Peterson, *Petri Nets Theory and the Modelling of Systems*, Prentice Hall, 1983.
- ❑ D. Lea, *Concurrent Programming in Java — Design principles and Patterns*, The Java Series, Addison-Wesley, 1996.

Petri nets: a definition

A *Petri net* C = ⟨P,T,I,O⟩ consists of:

1. A finite set P of *places*
2. A finite set T of *transitions*
3. An *input* function I: T $\rightarrow \mathcal{N}^P$ (maps to *bags* of places)
4. An *output* function O: T $\rightarrow \mathcal{N}^P$

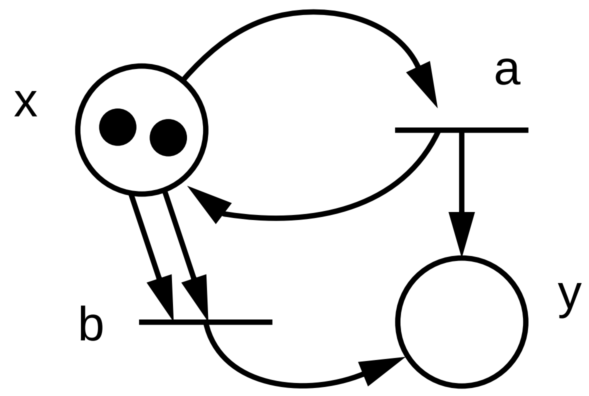
A *marking* of C is a mapping μ: P $\rightarrow \mathcal{N}$

Example:

P = { x, y }
T = { a, b }

I(a) = { x },	I(b) = { x, x }
O(a) = { x, y },	O(b) = { y }

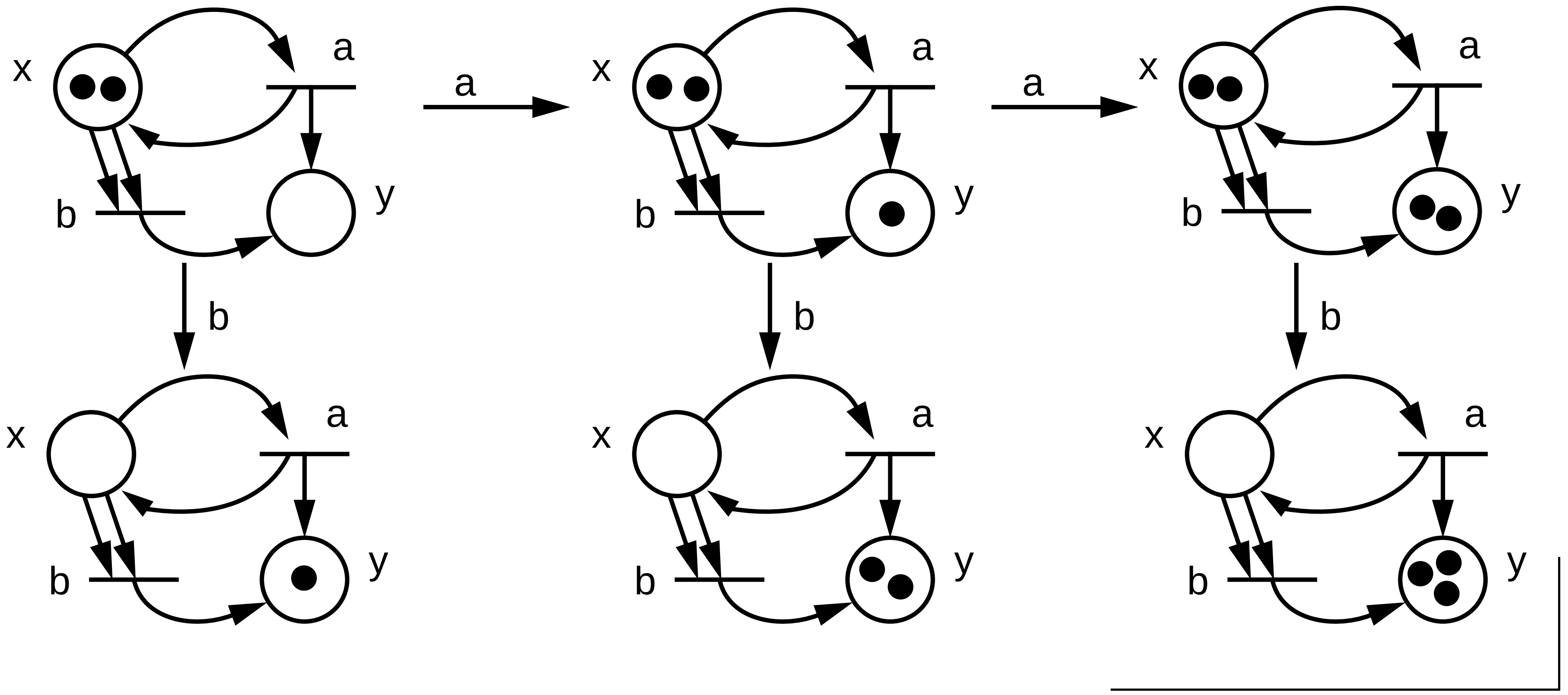
μ = { x, x }



Firing transitions

To fire a transition t:

1. There must be enough input tokens: $\mu \geq I(t)$
2. Consume inputs and generate output: $\mu' = \mu - I(t) + O(t)$



Modelling with Petri nets

Petri nets are good for modelling:

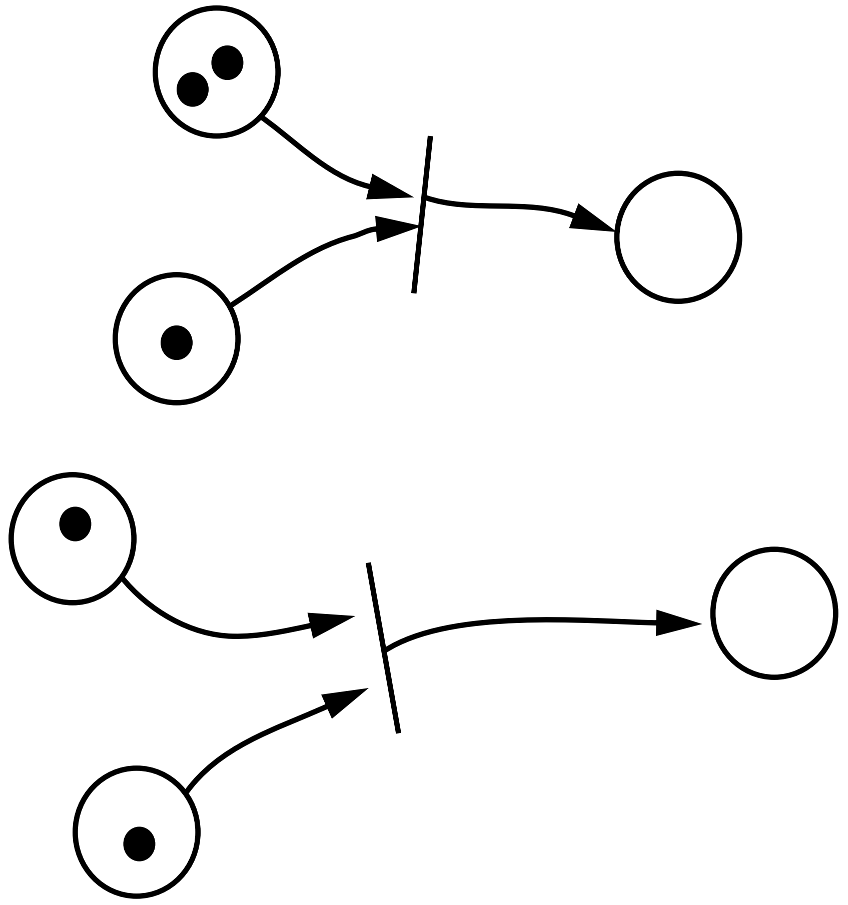
- concurrency
- synchronization

Tokens can represent:

- resource availability
- jobs to perform
- flow of control
- synchronization conditions ...

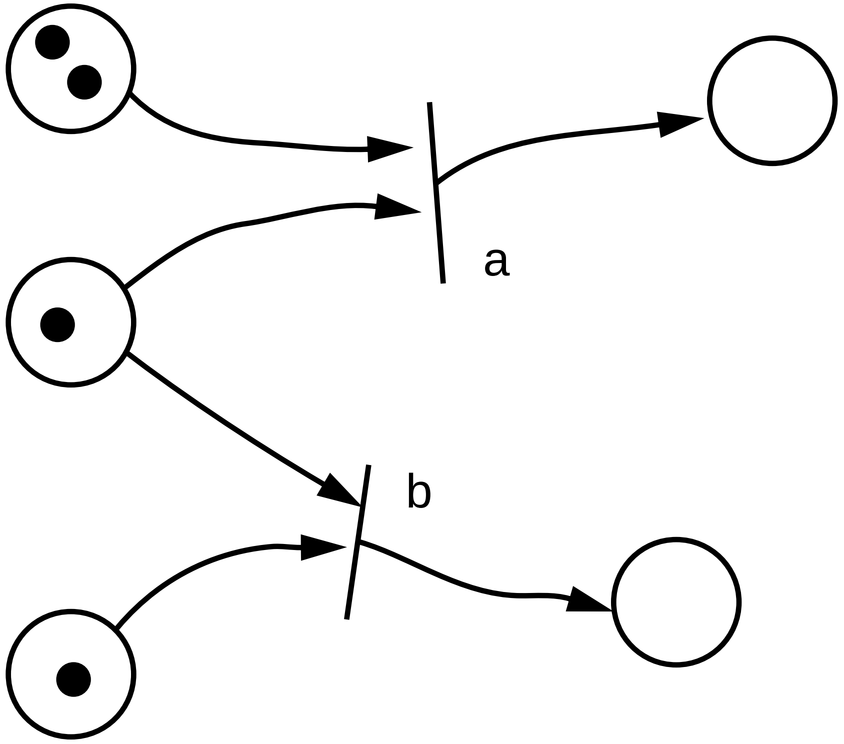
Concurrency

Independent inputs permit “concurrent” firing of transitions



Conflict

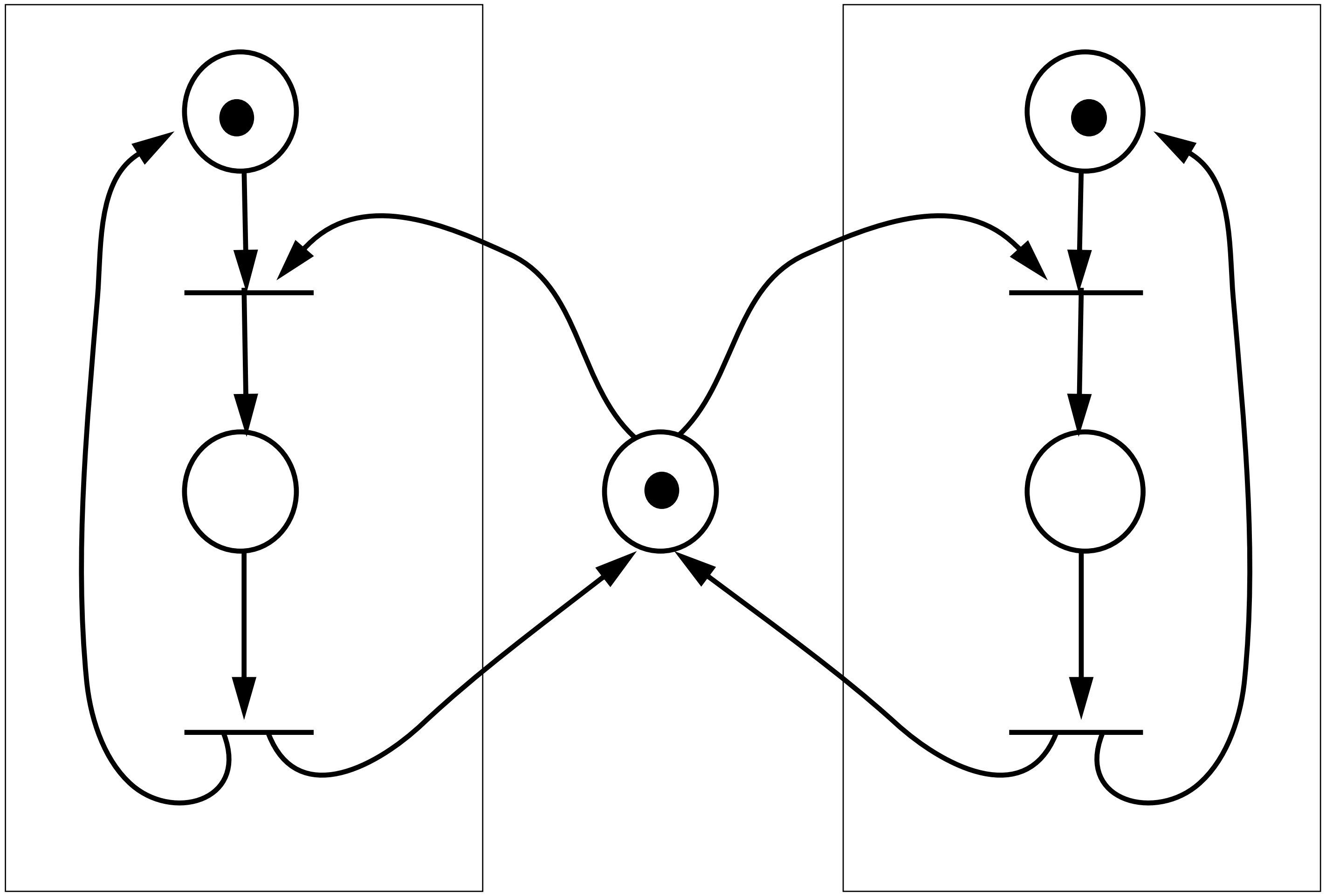
Overlapping inputs put transitions in conflict



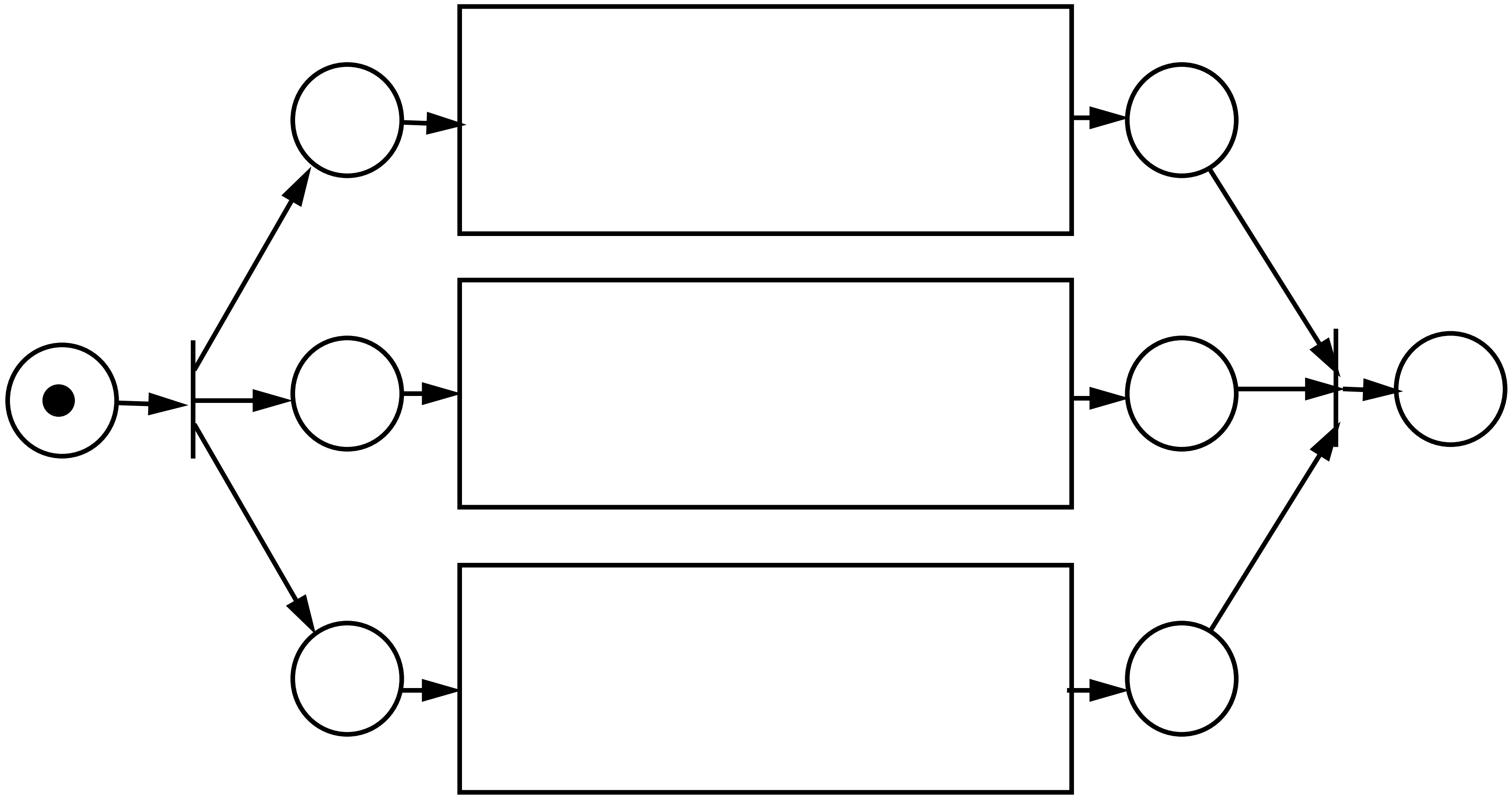
Only one of a or b may fire

Mutual Exclusion

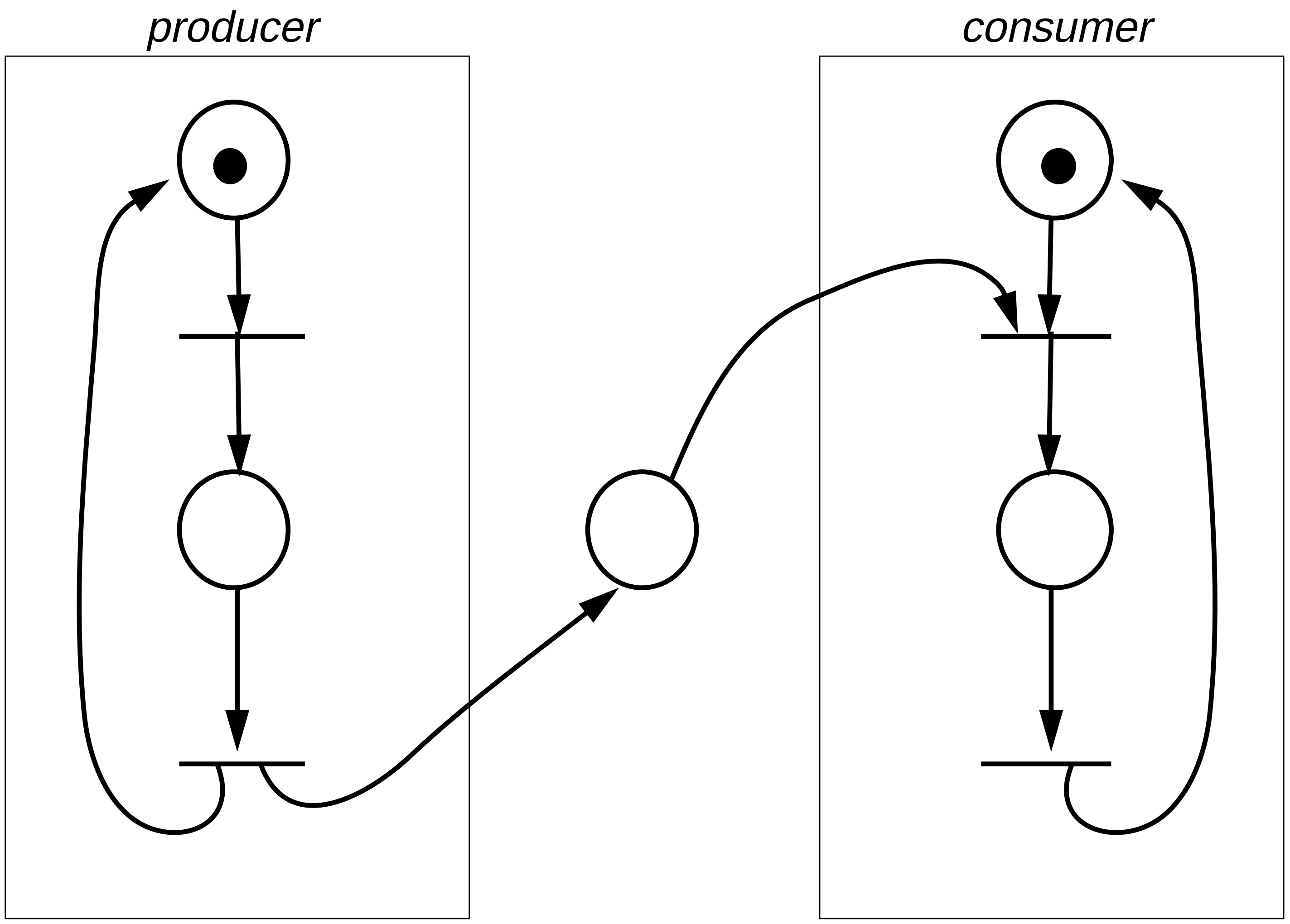
The two subnets are forced to synchronize



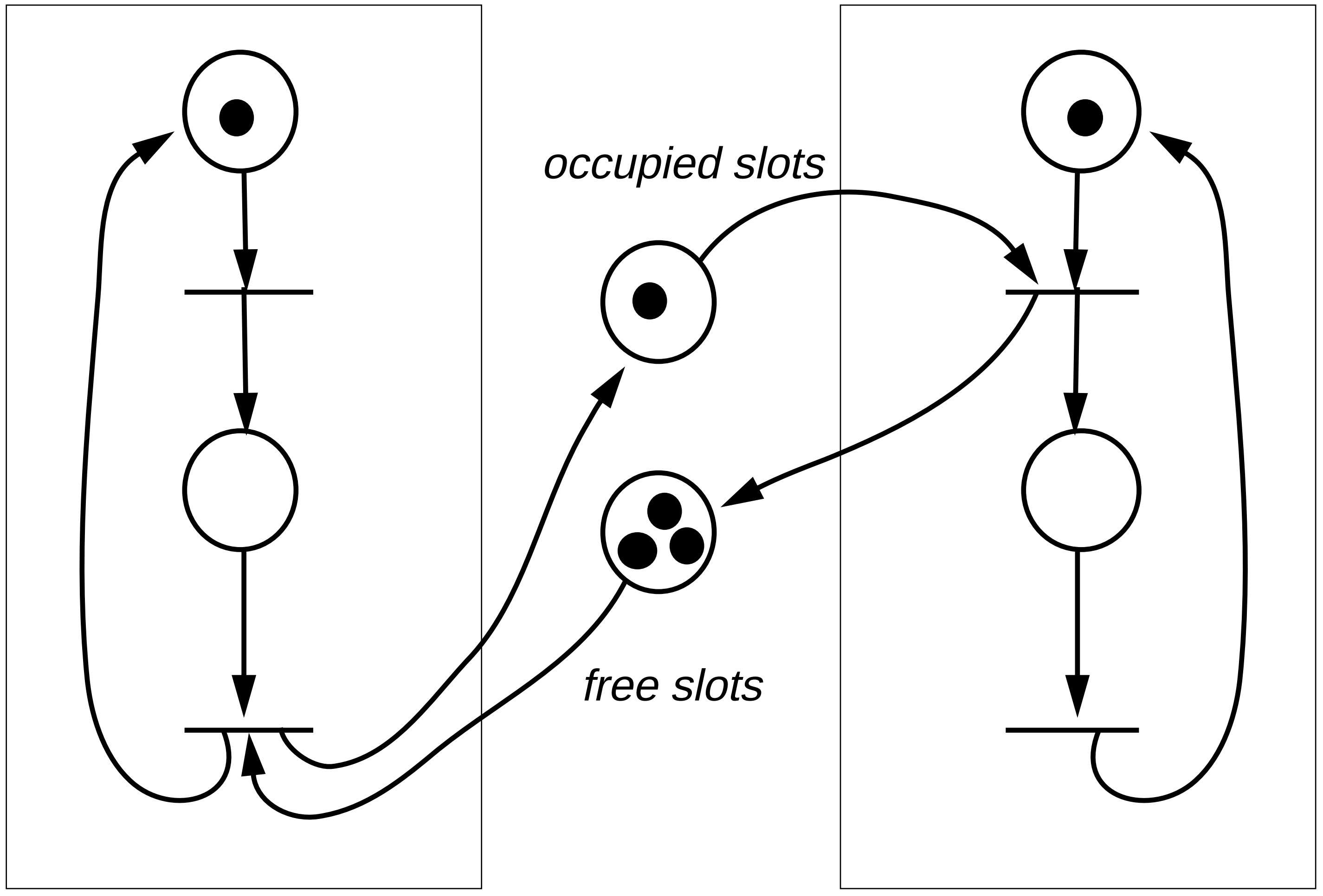
Fork and Join



Producers and Consumers



Bounded Buffers



Properties

Reachability:

- ❑ The *reachability set* R(C,μ) of a net C is the set of all markings μ' reachable from initial marking μ.

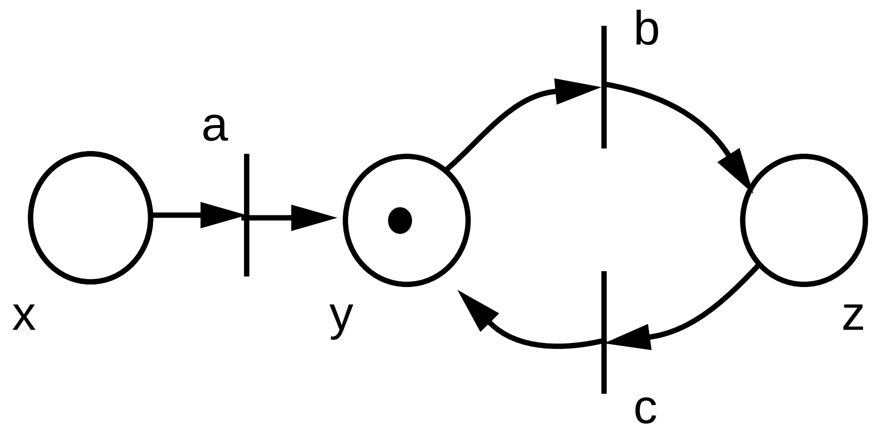
Boundedness:

- ❑ A net C is *safe* if places always hold at most 1 token.
- ❑ A net is *(k-)bounded* if places never hold more than k tokens.
- ❑ A net is *conservative* if the number of tokens is constant.

Liveness:

- ❑ A transition is *deadlocked* if it can never fire.
- ❑ A transition is *live* if it can never deadlock.

Liveness and Boundedness



This net is both *safe* and *conservative*.

Transition a is *deadlocked.*

Transitions b and c are both *live*.

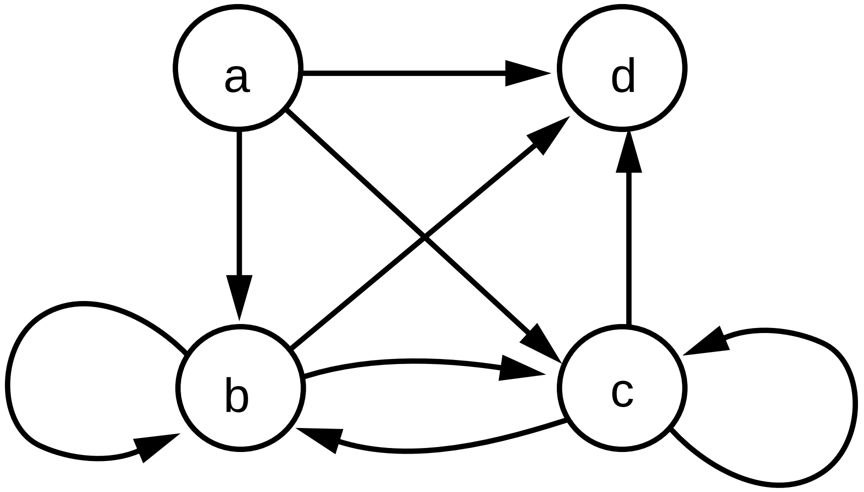
The reachability set is {{y}, {z}}.

Related Models

Finite State Automata

- Equivalent to *regular expressions*
- Can be modelled by one-token conservative nets
- Cannot model unbounded Petri nets

The FSA for: a(b|c)*d



Computational Power

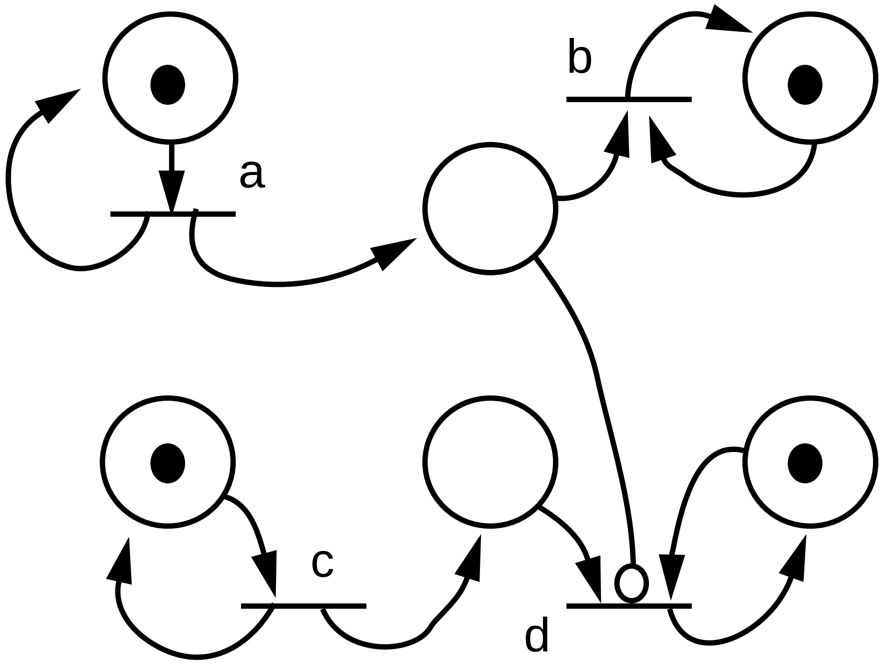
Petri nets are not computationally complete

- ❑ Cannot model "zero testing"
- ❑ Cannot model priorities

A zero-testing net:

An equal number of
a and b transitions may fire
as a sequence during any
sequence of matching
c and d transitions.

(#a ≥ #b, #c ≥ #d)



Applications of Petri nets

Modelling information systems:

- Workflow
- Hypertext *(possible transitions)*
- Dynamic aspects of OODB design

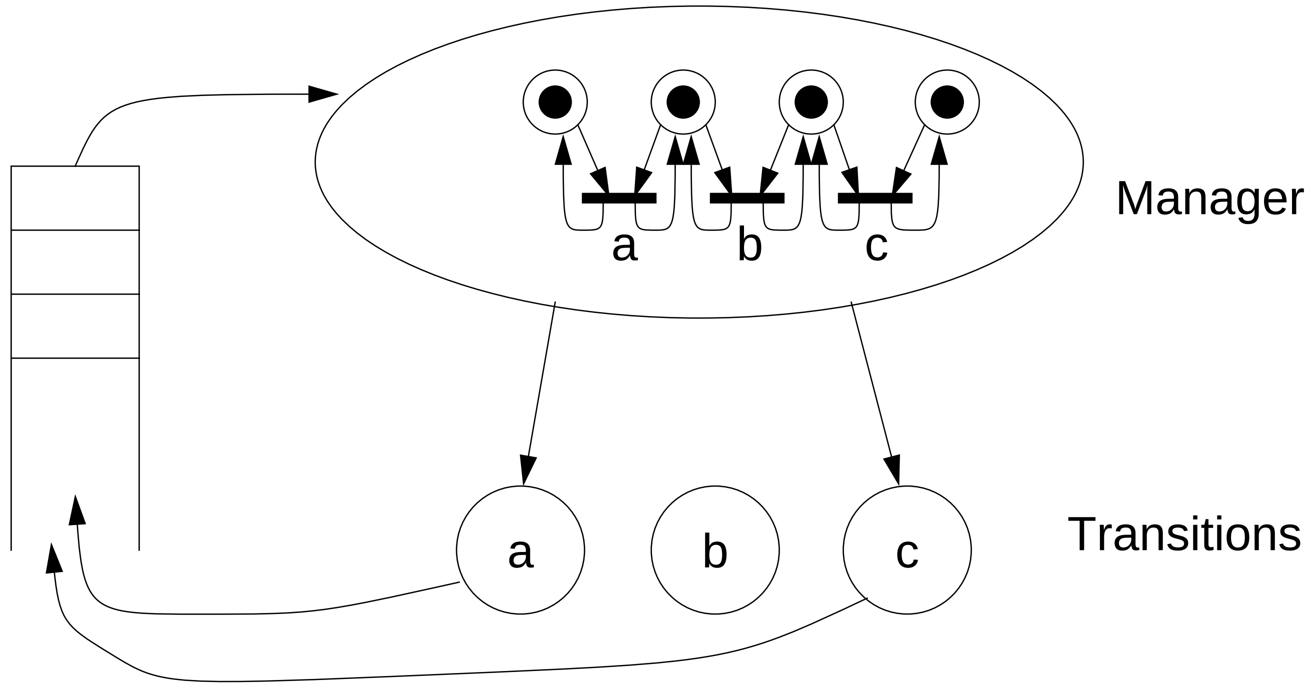
Implementing Petri nets

We can implement Petri net structures in either centralized or decentralized fashion:

- ❑ Centralized:
  - ☞ A single “net manager” monitors the current state of the net, and fires enabled transitions.
- ❑ Decentralized:
  - ☞ Transitions are processes, places are shared resources, and transitions compete to obtain tokens.

Centralized schemes

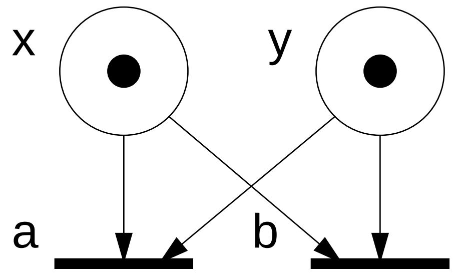
In one possible centralized scheme, the Manager selects and fires enabled transitions. When no transitions are enabled, it waits for tokens to be returned to its input queue:

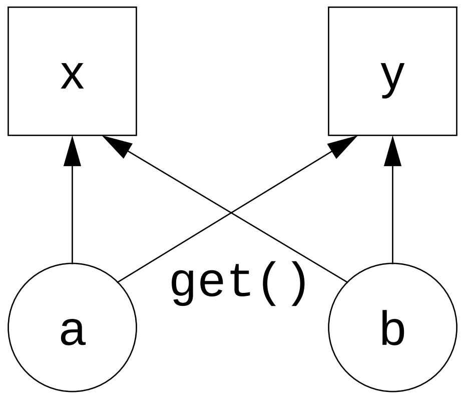


✎ *What liveness problems can this scheme lead to?*

Decentralized schemes

In decentralized schemes transitions are processes and tokens are resources held by places:



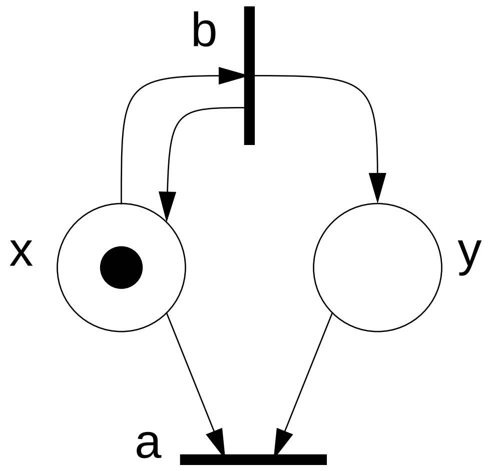


Transitions can be implemented as thread-per-message gateways so the same transition can be fired more than once if enough tokens are available.

Tokens must be grabbed in a consistent order, or the net can deadlock even though transitions are enabled!

Transactions

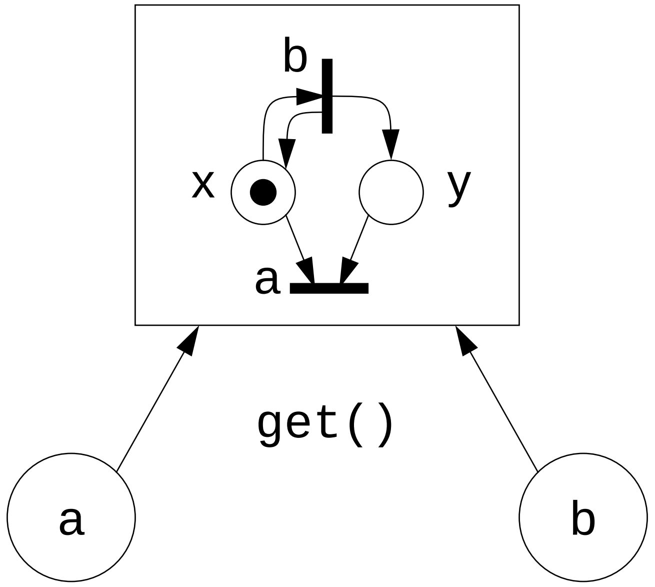
Transitions attempting to fire must grab their input tokens as an atomic transaction, or the net may deadlock even though there are enabled transitions:



If a and b are implemented by independent processes, and x and y by shared resources, this net can deadlock even though b is enabled if a (incorrectly) grabs x and waits for y.

Coordinated interaction

A simple solution is to treat the state of the entire net as a single, shared resource:



If a transition is not enabled, it waits and releases the net till it changes state again. When a transition fires and updates the net, it notifies all waiting transitions.

✎ *How could you refine this scheme to work in a distributed setting?*

Summary

You Should Know The Answers To These Questions:

- ❑ How are Petri nets formally specified?
- ❑ How can nets model concurrency and synchronization?
- ❑ What is the “reachability set” of a net? How can you compute this set?
- ❑ What kinds of Petri nets can be modelled by finite state automata?
- ❑ How can a (bad) implementation of a Petri net deadlock even though there are enabled transitions?
- ❑ If you implement a Petri net model, why is it a good idea to realize transitions as “gateways”?

Can You Answer The Following Questions?

- ✎ *What are some simple conditions for guaranteeing that a net is bounded?*
- ✎ *How would you model the Dining Philosophers problem as a Petri net?*
  Is such a net bounded? Is it conservative? Is it live?
- ✎ *What could you add to Petri nets to make them Turing-complete?*
- ✎ *What constraints could you put on a Petri net to make it fair?*

11. The pi Calculus

Overview

Introduction

- The π-calculus is a model of concurrent computation based upon the notion of *naming*.
- The π-calculus is a calculus in which the topology of communication can evolve dynamically during evaluation.
- In the π-calculus communication links are identified by *names*, and computation is represented purely as the communication of names across links.
- The π-calculus is an extension of the process algebra CCS, following work by Engberg and Nielsen who added mobility to CCS while preserving its algebraic properties.
- The most popular versions of the π-calculus are the monadic π-calculus, the polyadic π-calculus, and the simplified polyadic π-calculus.

Basic ideas

The most primitive entity in the π-calculus is a *name*. Names, infinitely many, are x, y, ... $\in N$; they have no structure. In the basic version of the π-calculus we only have one other kind of entity: a *process*. Processes are P, Q, ... $\in Pr$ and build from names by the following syntax:

$$Pr ::= \Sigma_{i \in \boldsymbol{I}}\, \pi_i.P_i \;\Big|\; P|Q \;\Big|\; !P \;\Big|\; (\upsilon\, x)P$$

Here $\boldsymbol{I}$ is a finite indexing set; in the case $\boldsymbol{I} = \emptyset$ we write the sum as $\mathbf{0}$. In a summand $\pi.P$ the prefix π represents an *atomic action*, the first action performed by $\pi.P$. There are two basic forms of prefix:

$x(y)$, which binds y in the prefixed process, means

"input some name - call it y - along the link named x",

$\overline{x}y$, which does not bind y, means

"output the name y along the link named x".

In each case we call x the subject and y the object of an action.

Simple examples

$$\overline{x}y.0 \mid x(u).\overline{u}v.0 \mid \overline{x}z.0$$

can evolve to

$$0 \mid \overline{y}v.0 \mid \overline{x}z.0 \text{ or } \overline{x}y.0 \mid \overline{z}v.0 \mid 0$$

$$(\upsilon\, x)(\overline{x}y.0 \mid x(u).\overline{u}v.0) \mid \overline{x}z.0$$

evolve to

$$0 \mid \overline{y}v.0 \mid \overline{x}z.0$$

$$\overline{x}y.0 \mid !x(u).\overline{u}v.0 \mid \overline{x}z.0$$

can evolve to

$$0 \mid \overline{y}v.0 \mid !x(u).\overline{u}v.0 \mid \overline{x}z.0 \text{ or } \overline{x}y.0 \mid \overline{z}v.0 \mid !x(u).\overline{u}v.0 \mid 0$$

and

$$0 \mid \overline{y}v.0 \mid !x(u).\overline{u}v.0 \mid \overline{z}v.0 \mid 0$$

The polyadic pi calculus

$P, Q ::= P \mid P$	*Parallel composition*
$(\upsilon\, x)\, P$	*Restriction*
$P + P$	*Summation*
$x[x_1, \ldots, x_n].P$	*Input*
$\overline{x}[x_1, \ldots, x_n].P$	*Output*
$!P$	*Replication*
$\mathbf{0}$	*Null*

$$!P \equiv P \mid\ !P$$

$$P \equiv P \mid \mathbf{0} \qquad P \equiv P + \mathbf{0}$$

$$P \mid Q \equiv Q \mid P \qquad P + Q \equiv Q + P$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(P + Q) + R \equiv P + (Q + R)$$

$$(\upsilon\, x)P \mid Q \equiv (\upsilon\, x)(P \mid Q) \quad x \notin \mathrm{fv}(Q)$$

$$\frac{Q \rightarrow R}{P \mid Q \rightarrow P \mid R} \qquad \frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q \equiv Q'}{P \rightarrow Q} \qquad \frac{P \rightarrow Q}{(\upsilon\, x)P \rightarrow (\upsilon\, x)Q}$$

$$(P + c[x_1, \ldots, x_n].Q) \mid (\overline{c}[y_1, \ldots, y_n].R + S) \rightarrow \{ x_1, \ldots, x_n \setminus y_1, \ldots, y_n \}Q \mid R$$

Church’s encoding of boolean in pi

λ:

True = $\lambda true.\lambda false.true$
False = $\lambda true.\lambda false.false$
Not = $\lambda arg.\lambda true.\lambda false.arg\ false\ true$

Not True = $(\lambda arg.\lambda true.\lambda false.arg\ false\ true)\ \lambda true.\lambda false.true$
$\rightarrow \lambda true.\lambda false.(\lambda true'.\lambda false'.true')\ false\ true$
$\rightarrow \lambda true.\lambda false.false$
= **False**

π:

True(b) = $b(t, f).\bar{t}$
False(b) = $b(t, f).\bar{f}$
Not(b, c) = $b(t, f).\bar{c}(f, t)$

$(\upsilon\ c)($**Not**(b, c) | **True**$(c))$ = **False**(b)
$(\upsilon\ c)(\ b(t, f).\bar{c}(f, t) \mid c(t, f).\bar{t}\ ) = b(t, f).\bar{f}$

?

Observable equivalence

Definition:

Two systems are equivalent whenever by interacting with them from the outside world, no difference can be observed.

Observation predicate:

A process P is observable at α, written $P{\downarrow_\alpha}$, if some $\alpha.A$ occurs unguarded in P.

if $P =^{def} \overline{a}[y] + b[x] + \tau.c$

then we have $\{ z : P{\downarrow_z} \} = \{ a, b \}$ and $\{ z : P{\Downarrow_z} \} = \{ a, b, c \}$

Definition:

(Strong) reduction equivalence, $\stackrel{\bullet}{\sim}_r$, is the largest equivalence relation $\models$ over processes such that $P \models Q$ implies

1. If $P \rightarrow P'$, then $Q \rightarrow Q'$ for some Q' such that $P' \models Q'$.
2. For each a, if $P{\downarrow_\alpha}$ then $Q{\downarrow_\alpha}$.

Observable equivalence II

Definition:

(Strong) reduction congruence, $\sim_r$, is the largest congruence relation over processes such that P $\sim_r$ Q implies that for all process contexts $\mathcal{C}[\,]$,

$\mathcal{C}[P] \overset{\bullet}{\sim}_r \mathcal{C}[Q]$.

A process context $\mathcal{C}[\,]$ is a process term with a single hole, such that placing a process in the hole yields a well-formed process.

True$(b) = b(t, f).\bar{t}$
False$(b) = b(t, f).\bar{f}$
Not$(b, c) = b(t, f).\bar{c}(f, t)$

$(\nu\, c)($**Not**(b, c) | **True**$(c)) \sim_r$ **False**(b)

$(\nu\, c)(\,b(t, f).\bar{c}(f, t) \mid c(t, f).\bar{t}\,) \sim_r b(t, f).\bar{f}$

Process typing

- In the π-calculus processes do not have types.
- Types are only assigned to names (channels).
- The type of a name (channel) remains constant throughout its lifetime.
- We do not specify temporal properties of names (channels).

Types:

$\delta ::= \hat{}[\,\delta_1, \ldots, \delta_n]$ Channel type

True$(b) = b(\,t, f\,).\bar{t} \;:\; \hat{}[\hat{}[], \hat{}[]]$

False$(b) = b(\,t, f\,).\bar{f} \;:\; \hat{}[\hat{}[], \hat{}[]]$

Not$(b, c) = b(\,t, f\,).\bar{c}(f, t) \;:\; \hat{}[\hat{}[], \hat{}[]]$

A simplification - polyadic mini pi-calculus

$P, Q ::= P \mid P$	*Parallel composition*
$(\upsilon\ x)\ P$	*Restriction*
$x[x_1, \ldots, x_n].P$	*Input*
$\overline{x}[x_1, \ldots, x_n]$	*Output*
$!P$	*Replication (Input)*
$\mathbf{0}$	*Null*

$$!P \equiv P \mid\ !P$$

$$P \equiv P \mid \mathbf{0} \qquad P \mid Q \equiv Q \mid P$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\upsilon\ x)P \mid Q \equiv (\upsilon\ x)(P \mid Q)\ \ x \notin \mathrm{fv}(Q)$$

$$\frac{Q \rightarrow R}{P \mid Q \rightarrow P \mid R} \qquad \frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q \equiv Q'}{P \rightarrow Q} \qquad \frac{P \rightarrow Q}{(\upsilon\ x)P \rightarrow (\upsilon\ x)Q}$$

$$c[x_1, \ldots, x_n].Q \mid \overline{c}[y_1, \ldots, y_n].R \rightarrow \{\, x_1, \ldots, x_n \setminus y_1, \ldots, y_n \,\}Q \mid R$$

Objects in the pi-calculus

Sangiorgi’s translation of an untyped OC(Adabí/Cardelli) into the polyadic π-calculus:

$$[\![\{_{j\in 1..n}\, l_j = \zeta(y).b_j\}]\!]_p \;=^{def}\; \overline{p}[x].!x[l,r,y].(\; \Pi_{j\in 1..n}(l = l_j)\; [\![b_j]\!]_r\;)$$

$$[\![a.l_j]\!]_p \;=^{def}\; (\upsilon\; q)(\; [\![a]\!]_q \mid q[x].\overline{x}[l_j,p,x]\;)$$

$$[\![a.l_j \Leftarrow \zeta(y).b]\!]_p \;=^{def}\; (\upsilon\; q)([\![a]\!]_q \mid q[x].\overline{p}[x_{new}].!x_{new}[l,r,y].\; ((l = l_j)[\![b]\!]_r \mid (l \neq l_j)\overline{x}[l,r,y])\;)$$

$$[\![x]\!]_p \;=^{def}\; \overline{p}[x]$$

A basic object model in the pi calculus

!RefCell[init, result].

    (υ contents set get)

        ($\overline{\text{contents}}$[init]

        | $\overline{\text{result}}$[set, get]

        | !set[value, ack].contents[dummy].($\overline{\text{contents}}$[value] | $\overline{\text{ack}}$)

        | !get[result].contents[value].($\overline{\text{contents}}$[value] | $\overline{\text{result}}$[value]))

The πL-calculus

$A, B ::= A \mid A$	*Parallel composition*
$(\upsilon\ x)\ A$	*Restriction*
$x(X).A$	*Input*
$\overline{x}(F)$	*Output*
$!A$	*Replication (Input)*
$\mathbf{0}$	*Null*

$$F ::= X \mid \varepsilon \mid F\langle l = x\rangle$$

$$x, y, z ::= a \mid X_l$$

$$!A \equiv A \mid !A$$

$$A \equiv A \mid \mathbf{0} \qquad A \mid B \equiv B \mid A$$

$$(A \mid B) \mid C \equiv A \mid (B \mid C)$$

$$\varepsilon(X).A \equiv \mathbf{0} \qquad \overline{\varepsilon}(F) \equiv \mathbf{0}$$

$$(\upsilon\ x)P \mid Q \equiv (\upsilon\ x)(P \mid Q)\ x \notin \mathrm{fv}(Q)$$

$$\frac{A \rightarrow B}{C \mid A \rightarrow C \mid B} \qquad \frac{A \equiv A' \qquad A' \rightarrow B' \qquad B \equiv B'}{A \rightarrow B} \qquad \frac{A \rightarrow B}{(\upsilon\ x)A \rightarrow (\upsilon\ x)B}$$

$$c(X).A \mid \overline{c}(F) \rightarrow \{ x_1, \ldots, x_n \setminus y_1, \ldots, y_n \}A$$

PICT

Overview

- PICT core syntax
- Creating new channels
- Channel types
- Modelling language constructs
- A concurrent queue

Abstract Syntax of (Untyped) Core PICT

Proc =	*Val* **?** *Abs*
	Val **?*** *Abs*
	Val **!** *Val*
	Proc **|** *Proc*
	let new *Name* **in** *Proc* **end**
Abs =	*Pat* **>** *Proc*
Pat =	*Name*
	[*Pat* , ... **]**
	record *Id* = *Pat* , ... **end**
	Name @ *Pat*
	_

Val =	*Name*
	BasicVal
	[*Val* , ... **]**
	record end
	Val **with** *Id* **=** *Val* **end**
Name =	*Id*
BasicVal =	*String*

Binding Channels

All channel names must be bound, either by “let new” or by an input pattern:

```
run
        let new x in
                x![ ]
        |       (x?[ ]>print!"Got it!")
        end
```

NB: print is a built-in channel

Typed Channels

Channels in PICT are typed, and may only carry values matching their type:

```
Type =   ^ Type
         ! Type
         ? Type
         [ Type , ... ]
         Record end
         Type with Id : Type end
         Top
```

In most cases, types can be automatically inferred, and declarations are unnecessary:

```
run
    let new x : ^[ ] in
        x![ ]
    |   (x?[ ]>print!"Got it!")
    end
```

Synchrony and Asynchrony

Although PICT uses asynchronous message-passing, synchrony can be recovered by waiting for a response on a (fresh) channel:

```
def sem [p,v] >
            (p?r > r![ ])
      |     (v?*r > r![ ] | (p?r > r![ ]))
```

A definition is syntactic sugar for a (new) replicated process

```
let new sem
run (sem?*[p,v] >
            (p?r > r![ ])
      |     (v?*r > r![ ] | (p?r > r![ ])))
```

Note that all channel names are bound, and that channels can be passed as values.

Synchronizing Concurrent Clients

```
def client [p,v] >
        let new r, s1, s2 in
                p!r
        |       (r?[ ] > pr!["FIRST\n",s1])
        |       (s1?[ ] > pr!["SECOND\n",s2])
        |       (s2?[ ] > v!r | (r?[ ] > skip))
        end

run
        let new p, v in
                sem![p,v]
        |       client![p,v]
        |       client![p,v]
        |       client![p,v]
        end
```

Modelling Booleans

```
def tt [b] > b?*[t,_] > t![ ]
def ff [b] > b?*[_,f] > f![ ]

def test [b] >
        let new t, f in
                b![t,f]
        |       (t?[ ] > print!"True")
        |       (f?[ ] > print!"False")
        end

def notB [b,c] > c?*[t,f] > b![f,t]

run
        let new b, c in
                ff![b] | notB![b,c] | test![c]
        end
```

Modelling Language Constructs

Higher-level language constructs are modelled by translation to core PICT:

```
run
   let new x in
      x!false
   |  (x?b >
      if b
      then print!"True"
      else print!"False"
      end)
   end
```

is translated to:

```
run
   let new x in
      x!false
   |  (x?b >
      let new t,f in
         primif![b,t,f]
      |  (t?[ ] > print!"True")
      |  (f?[ ] > print!"False")
      end)
   end
```

Natural Numbers

A natural number n can be modelled by a channel *n* that reads a pair [p,z] of channels, and either sends z![ ] if it is equal to zero, or else sends p![k] where k represents n-1.

```
def zero [p,z] > z![ ]
def one [p,z] > p![zero]
def two [p,z] > p![one]
def three [p,z] > p![two]
def count [n] >
    let new p,z in
            n![p,z]
    |   (z?[ ] > print!"0")
    |   (p?[m] > print!"1+" | count![m])
    end

run count![three]
```

Counting

New numbers can be generated by constructing a *successor* process:

```
def succ [n, r] >
        let new s in
                r!s
        |       (s?*[p,z] > p![n])
        end

run
        let new r in
                succ![three,r]
        |       (r?s > count![s])
        end
```

Arithmetic

Arithmetic operators can be built up in the same way:

```
def add [m,n,r] >
        let new p, z in
                m![p,z]
        |       (z?[ ] > r!n)
        |       (p?[pm] >
                let new rn in
                        succ![n,rn]
                |       (rn?sn>add![pm,sn,r])
                end)
        end

run let new r in
                add![two,three,r]
        |       (r?s > count![s])
        end
```

Functional Notation

Infix notation and functional application are syntactic sugar for communication:

```
run printi!(2+5)
```

translates to:

```
run printi!((+)[2,5])
```

which translates to:

```
run
    let new r in
        (+)![2,5,r] | (r?value > printi!value)
    end
```

Functions as Processes

Functions can be defined as processes:

```
def double [n] = n+n
```

translates to:

```
def double [n,r] > r!(n+n)
```

which translates to:

```
def double [n,r] >
    let new r1 in
        (+)![n,n,r1]
    |   (r1?value > r!value)
    end

run printi!(double[5])
```

Functions as Processes

```
def fact [n] =
   if n == 0
   then 1
   else n * fact[n-1]
   end
```

translates to:

```
def fact [n,r] >
   let new br in
      (==)![n,0,br]
   |  (br?b >
      let new t, f in
         primif![b,t,f]
      |  (t?[ ] > r!1)
      |  (f?[ ] >
         let new nr in
            (-)![n,1,nr]
         |  (nr?k >
            let new kfr in
               fact![k,kfr]
            |  (kfr?kf >
               let new fr in
                  (*)![n,kf,fr]
               |  (fr?f > r!f)
               end)
            end)
         end)
      end)
   end
```

```
run printi!(fact[5])

120
```

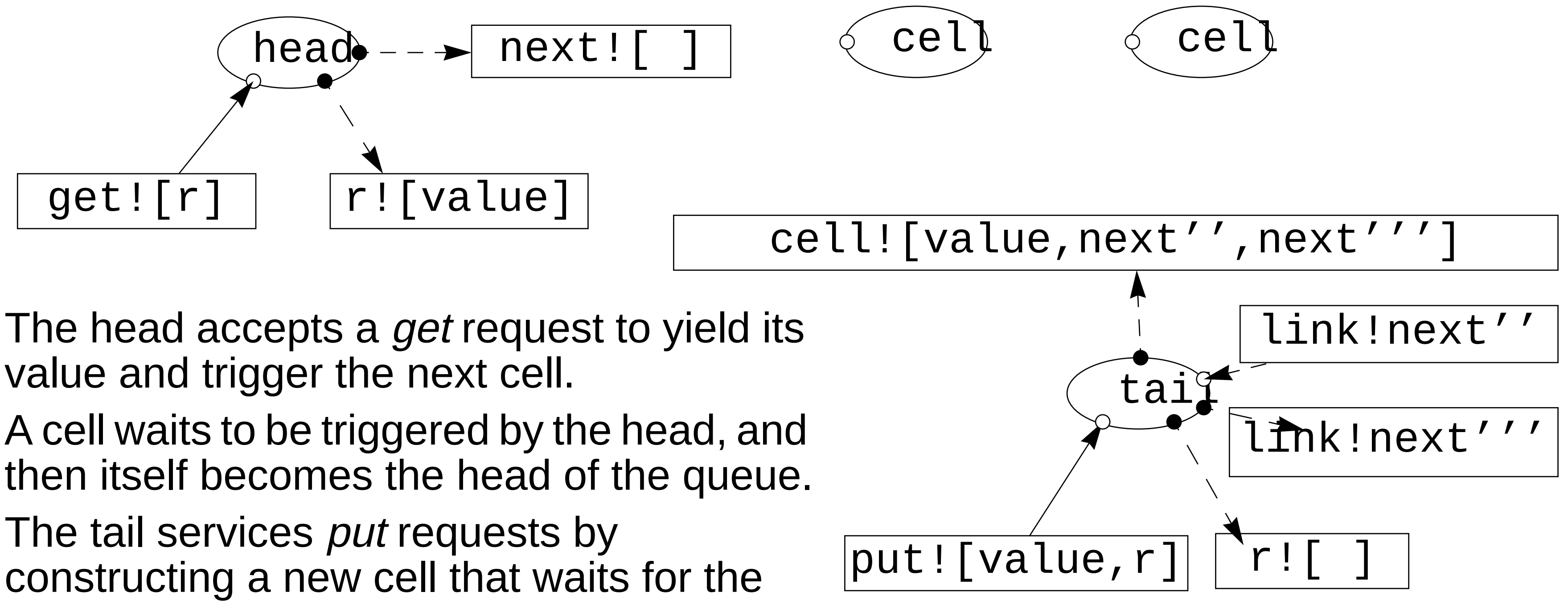
Sequencing

```
run
        pr["hello "];
        pr["world\n"];
        skip
```

translates to:

```
run
        let new r in
                pr!["hello ",r]
        |       (r?[ ] >
                let new r in
                        pr!["world\n",r]
                |       (r?[ ] > skip)
                end)
        end
```

A Concurrent Queue



The head accepts a *get* request to yield its value and trigger the next cell.

A cell waits to be triggered by the head, and then itself becomes the head of the queue.

The tail services *put* requests by constructing a new cell that waits for the *next* trigger from the cell in front of it.

Implementing the Concurrent Queue

```
new get, put


def head[value, next] >
   get?[r] > r!value | next![ ]


def cell[value, ready, next] >
   ready?[ ] > head![value, next]
```

```
def tail [ ] >
   let new link, init in
       link!init
   |  (put?*[value,r] >
       link?ready >
       let new next in
          cell![value,ready,next]
       |  link!next
       |  r![ ]
       end )
   |  init![ ]
   end
```

```
run
   let new r in
      tail![ ]
   |  (put["one"];  put["good"];  put["turn"];  put["deserves"];
put["another"]; skip)
   |  get![r]
   |  get![r]
   |  get![r]
   |  get![r]
   |  get![r]
   |  (r ?* s > print!s)
   end
```

12. JPict - the pi-Calculus in Java

Overview

Motivation

- The pi-calculus as a model for composition: $A \mid B$
- Implement pi-Model using Java-threads
- A *real* application of threads
- Notion of channels is extensible:
  - internal communication
  - user interaction
  - distributed communication: sockets
- Glue environment: A biotop for agents
  - E.g. Agents can listen on http-connections

The Architecture

Active elements (agents) are threads, that communicate by exchanging passive data (values) along channels.

1.	Parallel Agents:	$A \mid B$
2.	Restricted Agent:	new x A
3.	Sender	x ! y
4.	Receiver	x ? y > A
5.	Replicated Receiver	x ?* y > A

- ☞ a Java-thread executes an agent.
- ☞ replicated and parallel agents start new threads
- ☞ each agent or thread needs its own Environment, modelling the mapping from identifiers to values.

Synchronization

Synchronization is achieved by channels. A channel has `put()` and `get()` methods:

```
public class Channel {
   protected Vector queue;

   public synchronized Object get() {
      while (queue.size() == 0) {
         try { wait(); } catch (InterruptedException e) {} }
      Object v = queue.firstElement();
      queue.removeElementAt(0);
      return v;   // return head of queue

   }
   public synchronized void put(Object val) {
      queue.addElement(val); // add to tail of queue
      notify();
   }
}
```

✎ *What happens, if* `notifyall()` *would be used instead of* `notify()`

Here, we implement channels as FIFO-Buffers. We could also return a randomly chosen element from the queue in `get()`.

Running Agents

A running agent needs a program (an agent) and an environment (instance Env). What are the synchronization issues?

- ☞ Agent is immutable (read only)
- ☞ Environment is written. We have to give each (new) thread a copy of the Environment.

```
public class Running implements Runnable {
   protected Agent p_ = null;
   protected Env e_ = null;
   public Running(Agent p, Env e) {
      Thread t = new Thread(this);
      p_ = p; e_ = e;
      t.start();
   }
   public void run() {
      Running r = this;
      while (r != null)
         r=r.p_.iter(r);
   }
}
```

Example: replicated Reader

```
public abstract class Agent {
   public abstract Running iter(Running r) throws Exception;
   // run an arbitrary Agent (from outside)
   public void run(Env e) {
      Running r = new Running(this, e);
   }
}
public class RepAgent {
   protected IdentToken chan_;
   protected IdentToken pattern_:
   protected Agent next_;
   public Running iter(Running r) {
      // get the channel
      Channel t = (Channel)r.getEnv().valueOfId(chan_);
      // get the Object
      Object v = t.get(r);
      run((Env)r.getEnv().clone());// start replicated Agent
      r.getEnv().bind(pattern_, v);// bind Value to Pattern_
      r.setA(next_);                // set next Agent
      return r;
   }
```

✎ *What would happen, if* `run()` *would be before* `t.get(r)`

No one is an island

We need agents to be able to communicate values with the rest of the world: extern channels are channels, where one part of the communicator is not an agent but (for example) an ordinary Java-Method:

```
public interface Extern {
   public void run(Object v) throws Exception;
}
```

```
public class Print implements Extern {
   public void run(Object v) {
      System.out.println(v);
   }
}
```

```
public class ExtChannel  extends StdChannel {
protected final String theclass_;
public synchronized void put(Object val) throws Exception {
   // load Class theclass_
   Class dest = Class.forName(theclass_);
   // create a new Instance
   Object o = dest.newInstance();
   // invoke this Instance
   ((Extern)o).run(val);
}
```

Environment

We can implement the environment as an ordinary Java Hashtable, which must be cloned. (Unfortunately. `java.utils.Hashtable` is not cloneable) This is reasonable to get a first running version. But it is tedious slow.

There are two situations, when the environment must be cloned:

- Parallel Composition $A \mid B$
- Replicated Reading a ?* $x > A$

A look at the theory:

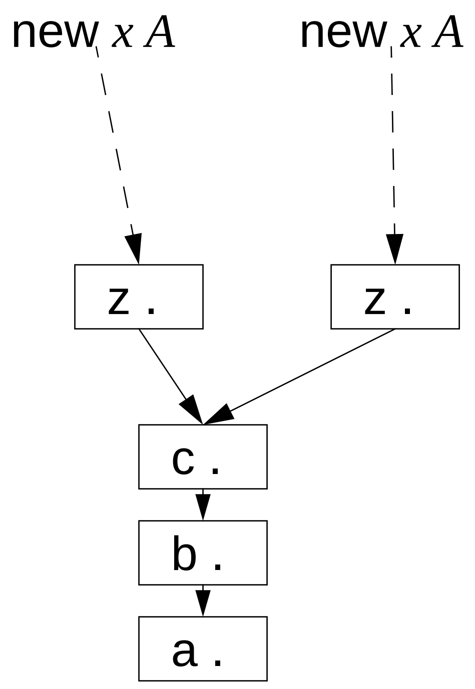
$$\{\text{new } x\ A\} \mid B \equiv \text{new x } \{A \mid B\} \text{ x} \notin \text{fv}(B)$$

this means, that $A \mid B$ can share the same environment, as long as they use distinct variables. This is easy, since variables are renamed by α-conversion. But we cannot move new across replicated agents:

$$a\ ?^*\ z > \{\text{new } x\ A\} \mid a\ !\ b \mid a\ !\ c \text{ ——> } a\ ?^*\ z \text{ new } x\ A \mid \text{new } x\ A \mid \text{new } x\ A$$

Environment II

Using α-renaming, we can derive that each Identifier is only bound once (before all reads) for a environment. We can model the environment as a tree, where value lookup start at the leaves:



- ❑ no need to clone the environment for a parallel agent
- ❑ no need to copy anything when replicating an agent.
- ❑ lookup length is calculated in advance
  - ➪ Performance is factors faster, but implementation needed some thought

Package jpict

a hierarchy of agents:

```
java.lang.Object
     |
     +----jpict.Agent
               |
               +----jpict.ChaAgent
               |         |
               |         +----jpict.ExtAgent
               |
               +----jpict.ParAgent
               |
               +----jpict.LocatedAgent
                         |
                         +----jpict.SndAgent
                         |
                         +----jpict.RecAgent
                         |
                         +----jpict.RepAgent
```

Lexer, Parser	8 Classes
Syntax Tree	15 Classes
Agents	11 Classes
Runtime	7 Classes
Exceptions	3 Classes

approx. 8’000 LOC (incl. comments)

Values

- What kind of values are exchanged via Channels

Name	*Values*
Monadic π-calculus	*Channels*
Polyadic π-calculus	*Tupels*
HOP-Calculi	*Agents / Abstractions*
πL	*Forms*
...	...

Translations exists between (some of) these different π-Versions. In JPict, we can send and receive `Objects` along `Channels`.

PiL - Forms

Forms are partial mappings from labels to channels (or Numbers, Strings). Forms can be extended yielding a new form, and projected against a label returning a channel.

y<reply = a>	A new form, y extended with a binding for label reply
y.reply	Denotes the value, bound by reply in form y

Forms behave much like the environments above. They are extended once, but read several times concurrently:

f ! *x<reply = b>* | *f* ! *x<reply = c>* | *f* ?* *y* > *y.reply* ! <>

Implementing Forms

This implementation yields immutable forms. We can forget about synchronization and sharing...

```
public class Form {
   private final Form prev_;// the predecessor Form (null for the empty Form)
   private final Object label_; // label of what I am extended from
   private final Object ch_; // Object bound by label_

   public Form() { prev_ = null; label_ = null; ch_ = null; }
   private Form(Form prev, Object label, Object ch) {
      prev_ = prev; label_= label; ch_ = ch;
   }

   public Object project(Object label) {
      if (label.equals(label_)) return ch_;
      else if (prev_ != null) return prev_.project(label);
      else return null; /* since we are the empty Form */
   }

   public Form extend(Object label, Object name) {
      return new Form(this, label, name);
   }
}
```

✎ *How would you implement mutable (extensible) Forms*

Modelling Boolean in PiL

A boolean is a modelled by a form, with labels `true` and `false`.

This agent receives a boolean and sends the empty form along the `true` channel of it. It sets the boolean to true:

```
b ? X > X.true ! <>
```

The True agent waits to get a channel b, which he can instantiate to true:

```
True ?* b > b.val ? X > X.true ! <>
```

An agent, testing a boolean b:

```
new truecase new falsecase
run { truecase ?* _ > print ! "it’s true” }
run { falsecase ?* _ > print ! “it’s false” }
run {b ! <true = truecase><false = falsecase>}
True ! b
```

But you can’t send Strings along Channels. The Translator reads: print ! <val = “it’s true”>

This agent swaps `true and false`:

```
Not ?* a > a.out ? X > a.in ! X<true = X.false><false = X.true>
```

Not True:

```
... new c { Not ! <in = c><out = b> | True ! c}
```

Extending the Boolean

A tree values logic has: true, false, unknown. not unknown = unknown.

We can reuse the encoding from the previous encoding. We only add a case for unknown:

```
Unknown ?* b > b.val ? X > X.unknown ! <>
```

and recode the test agent:

```
run {
   new truecase new falsecase new unknowncase
   run { truecase ?* _ > print ! "it’s still true" }
   run { falsecase ?* _ > print ! "it’s still false" }
   run { unknowncase ?* _ > print ! "it’s unknown" }

   // create the boolean channel
   new b
   new c

   b ! <true = truecase><false = falsecase><unknown = unknowncase>
   | Not ! <in = c><out = b>
   | Unknown! c
}
```

✎ *If we encoded* `Not` *as*
`Not ?* a > a.out ? X > a.in ! <true = X.false><false = X.true>`,
what would change?

Towards a programming language...

Often, we like to send an *agent abstraction* along a channel, where the receiver of the abstraction can invoke it:

This is an abstraction: when it receives a form a, it sends along `print` the value of `a.val`:

```
\a > print ! a.val


new x
run {
   x ! \a > print ! a.val
   x ? f > f ! <val = “A String”>
}
```

But, we can’t send abstractions. We can only send the location of an abstraction. The location is the channel, at which the agent listens:

```
new x
run {
   run { new f x ! f | f ?* a > print ! a.val }
   x ? f > f ! <val = “A String”>
}
```

✎ *Why is it necessary to use a replicated Agent at f?*

Functions, Assignment...

➪ Instead of communicating *complex things* we communicate (restricted) channels giving access to the thing.

	Syntax	*Mapped to*
Sending an Abstraction	*r ! \x > A*	*new f r ! <val = f> | f ?* x > A*
Function	*\x . E*	*\x > x.reply ! E*
Send the Result	*r ! f(x)*	*f ! x<reply = r>*
Assignment	*let x := E; A*	*new r r ! E | r ? x > A*

If *x*, *f* etc. is a form, we cannot send something along it. Write therefore *x.val ! E* instead of *x* ! *E*.

Example

```
extern "extern.Concat" ++
extern "extern.StdOut" pr
extern "extern.Subtraction" -
extern "extern.Smaller" lt
extern "extern.Addition" +

def fib x > if (x lt 3)
              then x.reply ! 1
              else x.reply ! fib(x - 2) + (fib(x - 1));


let result := fib(8);
pr(“It’s " ++ result);
```

Prints out: `It’s 21`

Although the code is purely sequential, the translation creates agents that wait on some channels, create new agents, and finally send something along another channel.

Some statistics: approx.: 800 ms

287 threads

➪ We should try to optimize this...

Java-Threads and Pi-Process

$$a \,!\, x \mid a \,?\, y > B$$

- Reader executes `get()` on channel a -> thread blocks
- Java thread context switch
- Sender executed `put()` -> notify of Reader
- Java thread context switch
- Reader receives value x and continues

Idea: reader stores his `Running` on the channel when getting a value and the channel is empty. The sender then returns this `Running` instance after `put()`, which is the natural continuation.

➪ channel is more complex
➪ No `wait(), notify()` needed in channel
➪ Fib(8) in 600 ms, but still as many threads

But we still have a lot of threads...

Java-Threads and Pi-Process II

The translation of higher order expression often contains a pattern like

$$\text{new } r\ A \mid r\ ?\ x > B$$

where A is an Agent sending a form along the reply channel r, and B is an agent doing something with the form received from r. Executing A and $r\ ?\ x > B$ in two threads is a waste, since the reader cannot proceed until A has really sent something along r.

Idea:

- `iter()` the reader agent. This just performs one `get()` on the (empty) channel
- then execute agent A.

So, instead of instantiating a parallel agent, we use a sequential agent, that does not start a new thread for $r\ ?\ x > B$

➪ needs a subclass of a parallel agent

➪ Fib(8) now in 400 ms, with 2 threads

Concurrent Queue in Pil

```
new get new put
def head x > { get ? y > { y ! <val = x> | x.next !<>}};
def cell x > x.ready ? _ > head ! <val = x><next = x.next>;
def tail _ > {
   new link new init run {link ! init}
   put ?* x > {
      link ? ready > {
         new next
         cell ! x<ready = ready><next = next>
         | link ! next
         | x.reply ! <>}
   }
   | init ! <>
};
run {
   new r
   tail ! <>
   | {put("one"); put("good"); put("turn"); put("deserves"); put("another");{}}
   | get ! r | get ! r | get ! r | get ! r | get ! r
   | r ?* x > print ! x
}
```

Uses 50 threads

Interested?

- Visualizing agents
  using Java-GUI/Applet techniques to *see* the agents at work
- Debugging (visually) running agents
- Using agents to script WWW-Servers
  viewing http-demons as agents that wait for some forms and return Information. Then build up an agent that communicates with these...
- ...