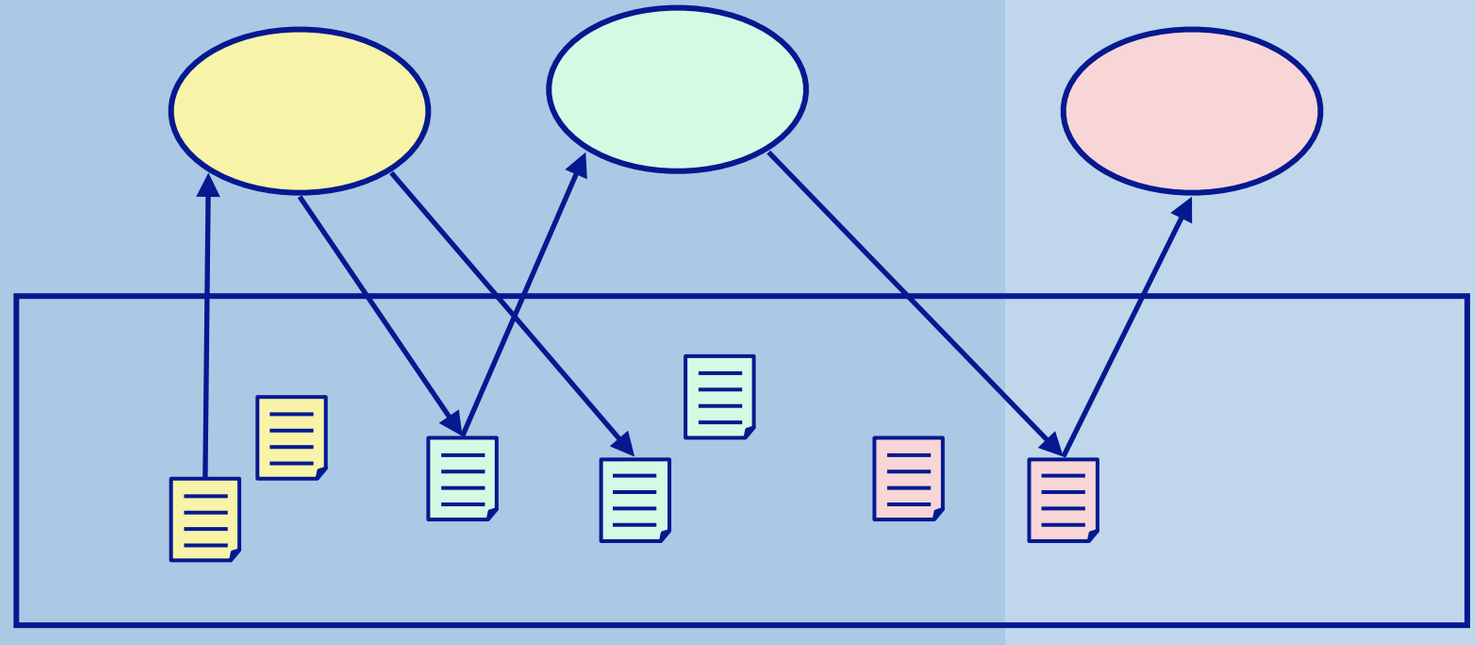
b
UNIVERSITÄT
BERN

12. Architectural Styles for Concurrency

Oscar Nierstrasz



Roadmap

- What is Software Architecture?
- Three-layered application architecture
- Flow architectures
  - —Active Prime Sieve
- Blackboard architectures
  - —Fibonacci with Linda

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.

The classic book by Shaw and Garlan introduces the notion of an “architectural style” being described in terms of "components” and "connectors". Buschmann's book describes a number of classical architectural design patterns. We have been following Lea’s book on concurrent design patterns in the course. Carriero and Gelernter describe how to design concurrent solutions using a “blackboard” architecture.

http://zoo.cs.yale.edu/classes/cs424/howto.pdf

http://scgresources.unibe.ch/Literature/CP/Carr89aSurvey.pdf

Roadmap

- **What is Software Architecture?**
- Three-layered application architecture
- Flow architectures
  - —Active Prime Sieve
- Blackboard architectures
  - —Fibonacci with Linda

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

Software architecture defines the coarse-level design of a software system in terms of large-scale “components” and how these components interact. E.g., a “fat client” architecture splits a system into a “fat” client that is responsible for as much user processing as possible, and a server that handles the actual requests. “Fat client” is an *architectural style* that applies to many systems.

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

A style introduces names for the kinds of components (eg “fat client”and “server”) and connectors, and constrains how they are composed and built. (E.g., the client and server have distinct responsibilities, and the possible interactions between them are limited.)

“Analyses” also include certain desirable properties that are guaranteed. With fat clients, users are guaranteed high response, since many requests are handled locally by the fat client.

Roadmap

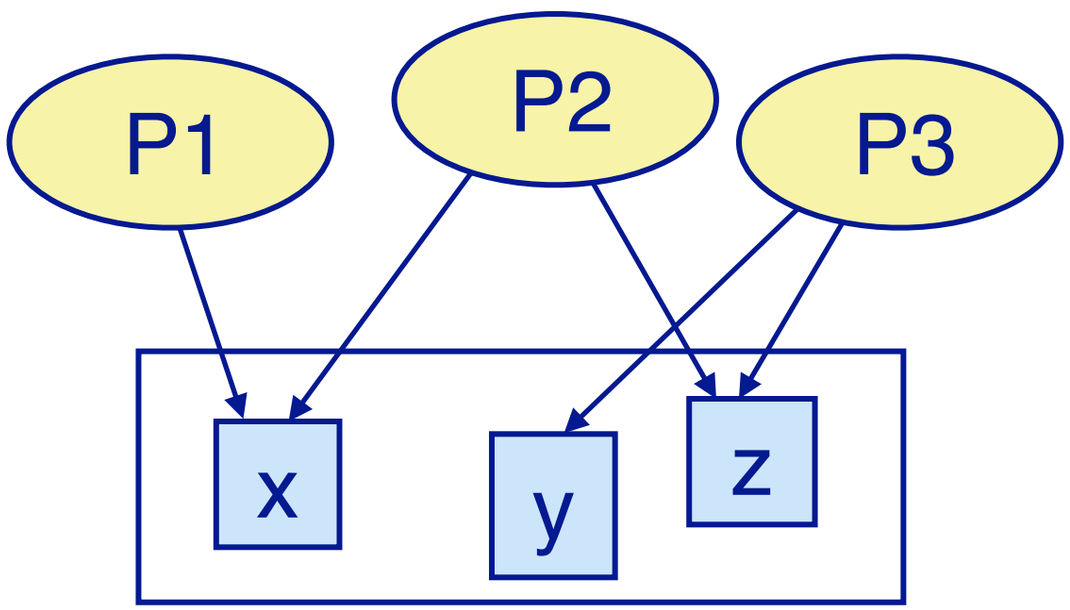
- What is Software Architecture?
- **Three-layered application architecture**
- Flow architectures
  - —Active Prime Sieve
- Blackboard architectures
  - —Fibonacci with Linda

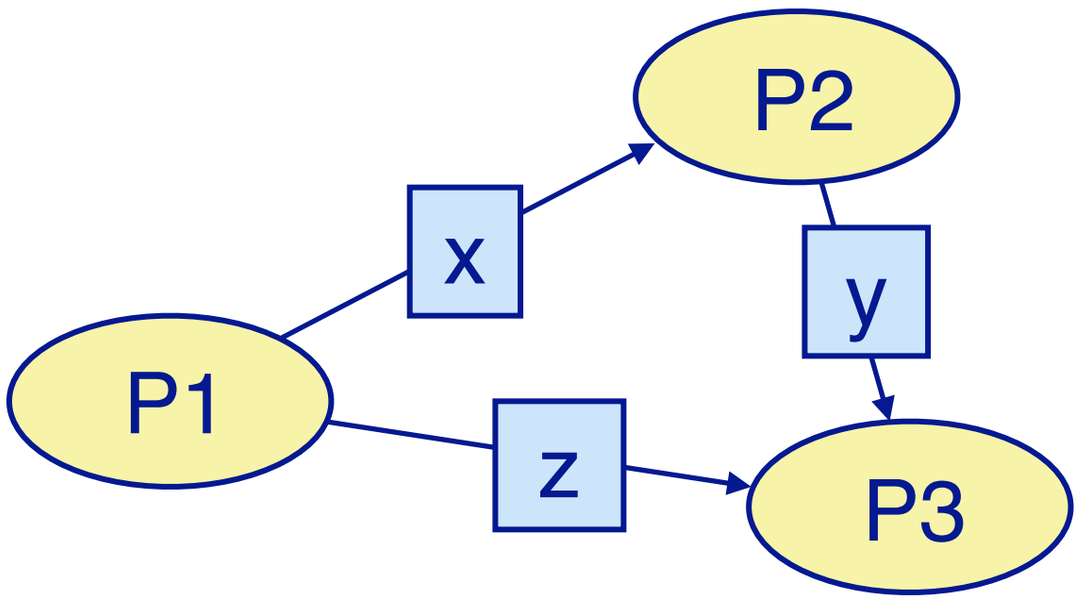
Communication Styles

Shared Variables

Processes communicate indirectly.

Explicit synchronization mechanisms are needed.





Message-Passing

Communication and synchronization are combined.

Communication may be either *synchronous or asynchronous*.

Recall these two basic styles of concurrent programming. In the first we require explicit synchronization of access to shared variables within critical sections. In the other, synchronization and communication are combined in the form of message passing.

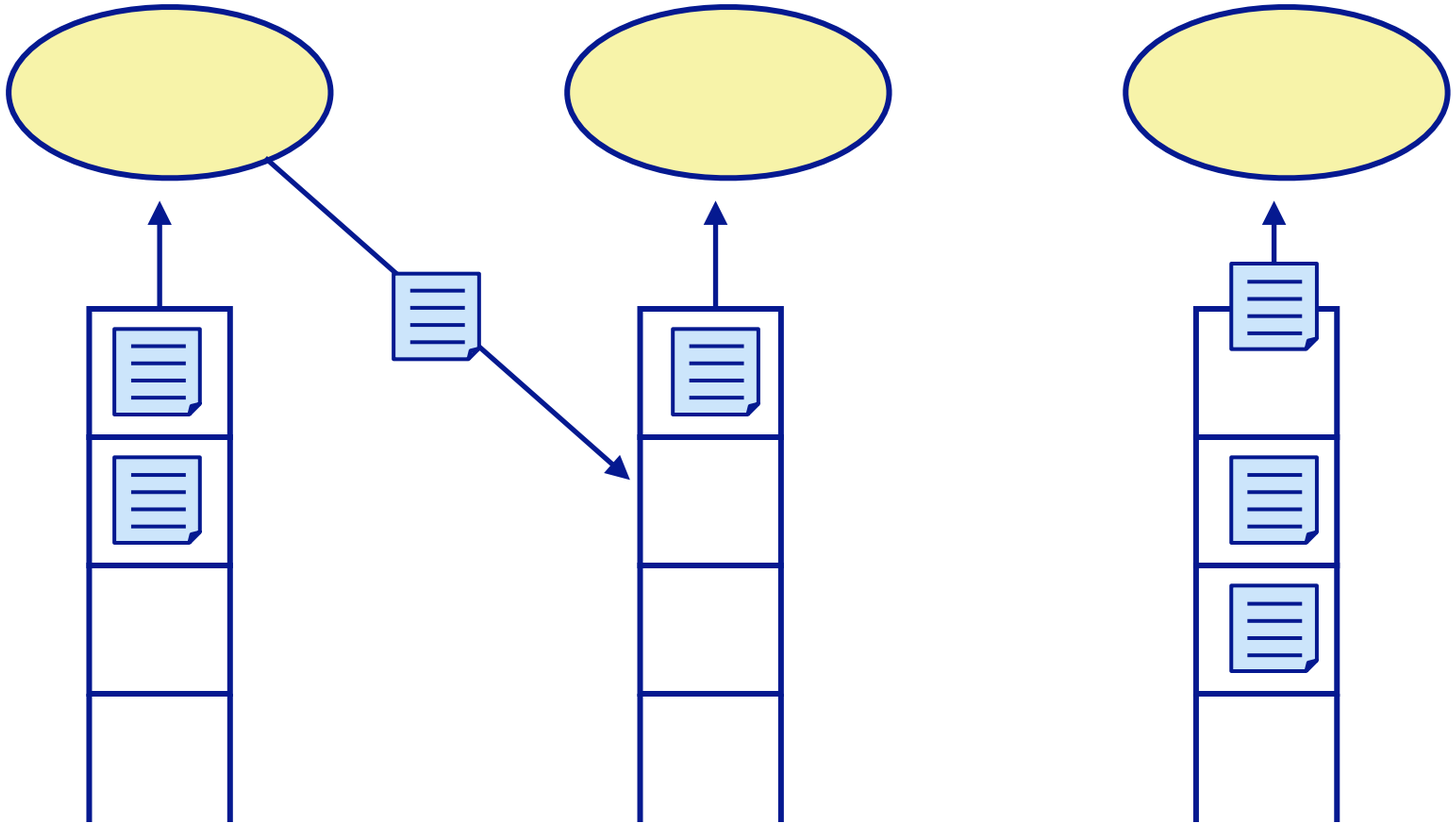
Simulated Message-Passing

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

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

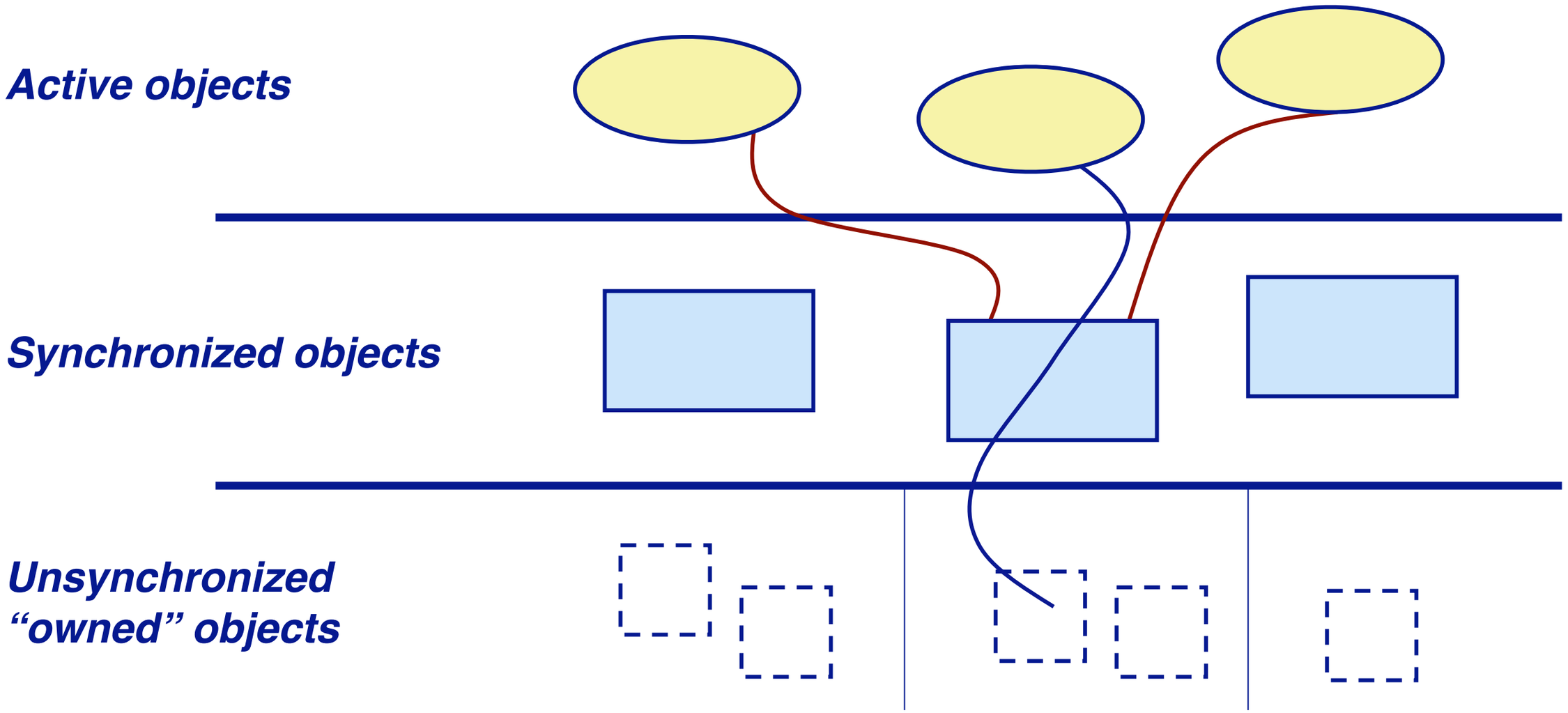
Unsynchronized objects

Synchronized queues



Message passing can be easily simulated by implementing message queues as synchronized, shared variables.

Three-layered Application Architectures



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

In a classical three-layered architecture we distinguish between “active objects” with their own threads of control, that communicate with each other through synchronized objects that themselves may contain further, unsynchronized objects. No nested monitor problems can arise since the synchronized objects never contain any monitors.

Note that message passing simulation follows this architectural style, with the messages in the queues corresponding to the third layer.

Roadmap

- What is Software Architecture?
- Three-layered application architecture
- **Flow architectures**
  - —Active Prime Sieve
- Blackboard architectures
  - —Fibonacci with Linda

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.

Unix Pipes

Unix pipes are *bounded buffers* that connect producer and consumer processes (*sources, sinks and filters*):

```
cat file                  # send file contents to output stream
| tr -c 'a-zA-Z' '\012' # put each word on one line
| sort                   # sort the words
| uniq -c                # count occurrences of each word
| sort -rn               # sort in reverse numerical order
| more                   # and display the result
```

This script counts the words in a text file using a pipeline of Unix commands. The command "`cat`" just concatenates the argument files and sends them to its output stream. This stream is read by the "`tr`" command, which performs simple translations. Here it translates non-alphabetic characters to a newline character, effectively putting each word on a separate line in the output. The remaining commands sort the list of words, count their occurrences, and sort them again from most to least common.

Unix Pipes

Processes should *read from standard input* and *write to standard output* streams:

—Misbehaving processes give rise to *“broken pipes”!*

Process creation and *scheduling* are handled by the O/S.

Synchronization is handled implicitly by the I/O system (through buffering).

A Unix command must obey the “pipeline contract” for it to be used in a pipes and filters pipeline. A “*source*” produces output only and must be the first command in the pipeline. A “*pipe*” reads the standard input stream and writes to standard output. It must read *all* its input, and not terminate prematurely. By default the last pipe sends its output to the terminal, but a “*sink*” can otherwise consume it, e.g., by producing an output file.

The operating system connects pipes with bounded buffers. A slow pipe will cause its neighbours to block as its input buffer is filled up and its output buffer becomes empty.

Flow Stages

Every flow stage is a *producer* or *consumer* or both:

> Splitters (Multiplexers) have *multiple successors*

—Multicasters *clone results* to multiple consumers

—Routers *distribute results* amongst consumers

> Mergers (Demultiplexers) have *multiple predecessors*

—Collectors *interleave inputs* to a single consumer

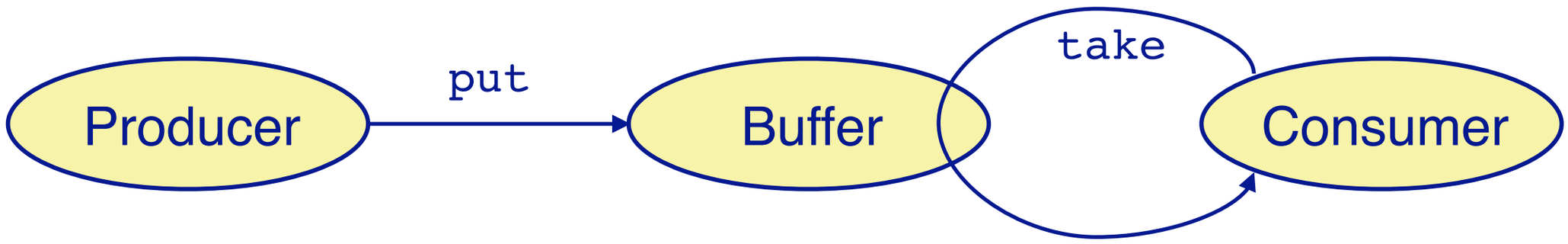
—Combiners *process multiple input* to produce a single result

> 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 (adapt) 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:

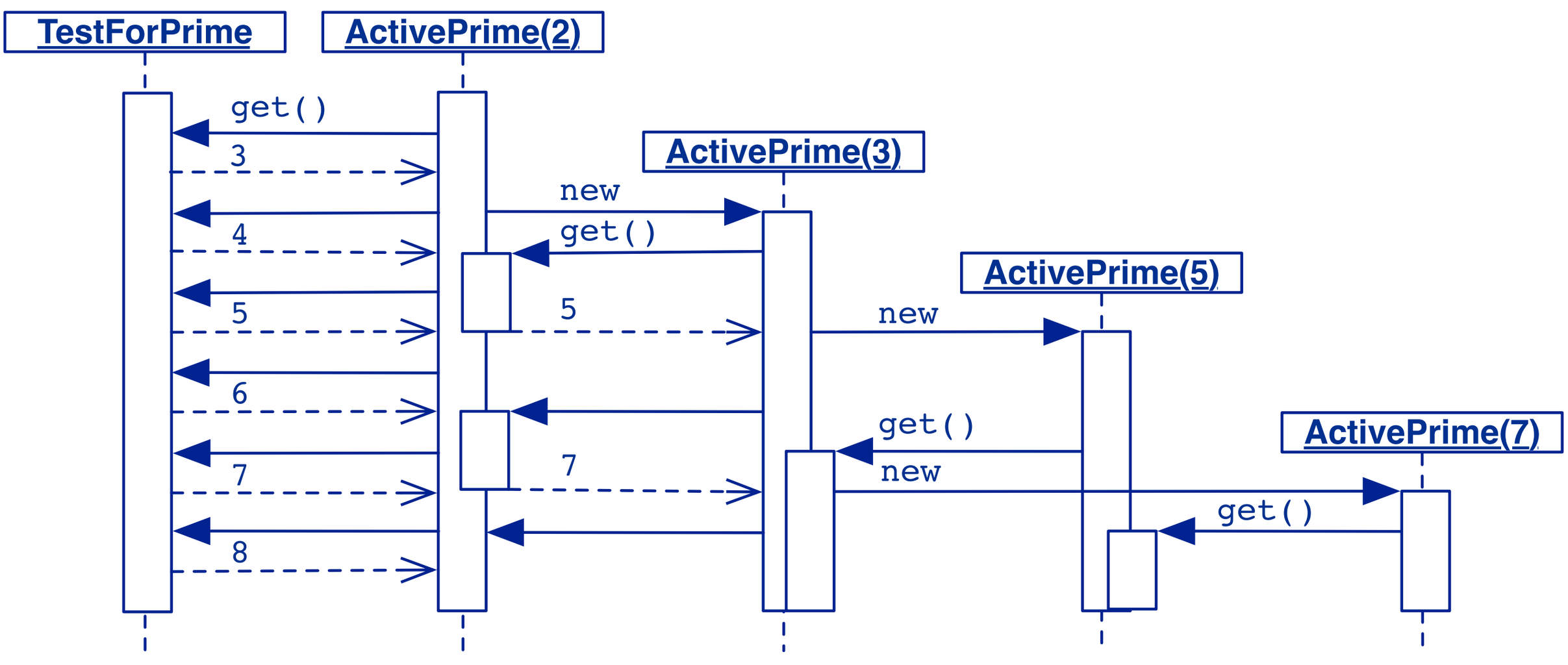
- *Harder to manage* than bounded buffers

Roadmap

- What is Software Architecture?
- Three-layered application architecture
- Flow architectures
  - **Active Prime Sieve**
- Blackboard architectures
  - Fibonacci with Linda

Example: a Pull-based Prime Sieve

Primes are agents that reject non-primes, pass on candidates, or instantiate new prime agents:

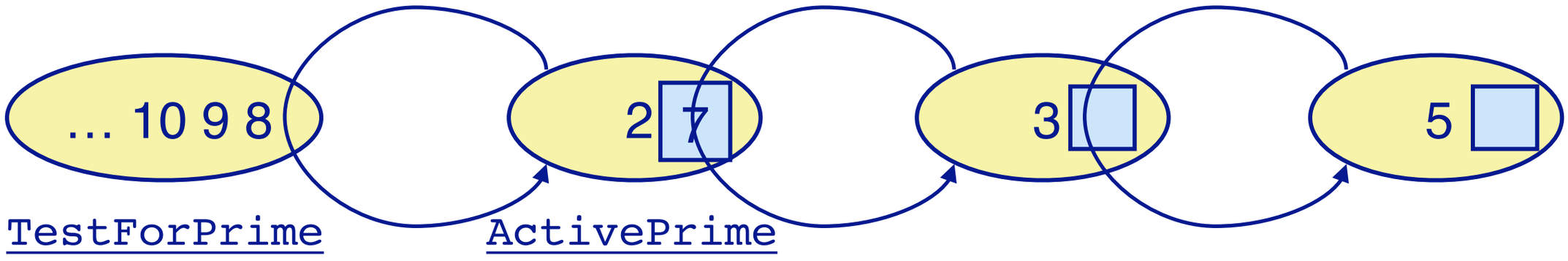


In this example, we dynamically build up a pipeline of `ActivePrime` objects, each of which holds a prime number, and tests an input stream of prime candidates for divisibility by the prime it holds. The `TestForPrime` source at the head of the pipeline produces a stream of integers. The pipeline is initialized only with `ActivePrime(2)`. When an `ActivePrime` finds candidate that passes its test, and is also smaller than the square of the prime, then it recognizes the candidate as a prime, and adds a new `ActivePrime` object to the end of the queue.

`ActivePrime(2)` thus promotes 3 to an `ActivePrime`, which in turn promotes 5 and 7 to `ActivePrimes`. ($3 < 4$, $5 < 9$ and $7 < 9$)

Using Put-Take Buffers

Each ActivePrime uses a one-slot buffer to feed values to the next ActivePrime.



The first ActivePrime holds the seed value 2, gets values from a TestForPrime, and creates new ActivePrime instances whenever it detects a prime value.

The PrimeSieve

The main PrimeSieve class creates the initial configuration

```
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) { }
   }
}
```

ActivePrimes

Pull-based integer sources

Active primes get values to test from a `Source<Integer>`:

```
public interface Source<Value> { Value get(); }
class TestForPrime implements Source<Integer> {
   private int nextValue;
   private int maxValue;
   public TestForPrime(int max) {
      this.nextValue = 3;
      this.maxValue = max;
   }
   public Integer get() {
      if (nextValue < maxValue) { return nextValue++; }
      else { return 0; }
   }
}
```

The ActivePrime Class

ActivePrimes themselves implement IntSource

```
class ActivePrime extends Thread implements Source<Integer> {
   private static Source<Integer> lastPrime; // shared
   private int value;                        // value of this prime
   private int square;                       // square of this prime
   private Source<Integer> intSrc;  // source of ints to test
   private OneSlotBuffer<Integer> slot;   // pass on test value
   public ActivePrime(int value, Source<Integer> intSrc)
      throws ActivePrimeFailure
   {
      this.value = value;
      ...
      slot = new OneSlotBuffer<Integer>();
      lastPrime = this;          // NB: set class variable
      this.start();
   }
```

The only synchronization is hidden within the Slot class.

Note that `lastPrime` is a shared variable updated by the `ActivePrime` constructor.

Why is it not necessary to synchronize access to this variable?

It is impossible for primes to be discovered out of order! Can you prove this?

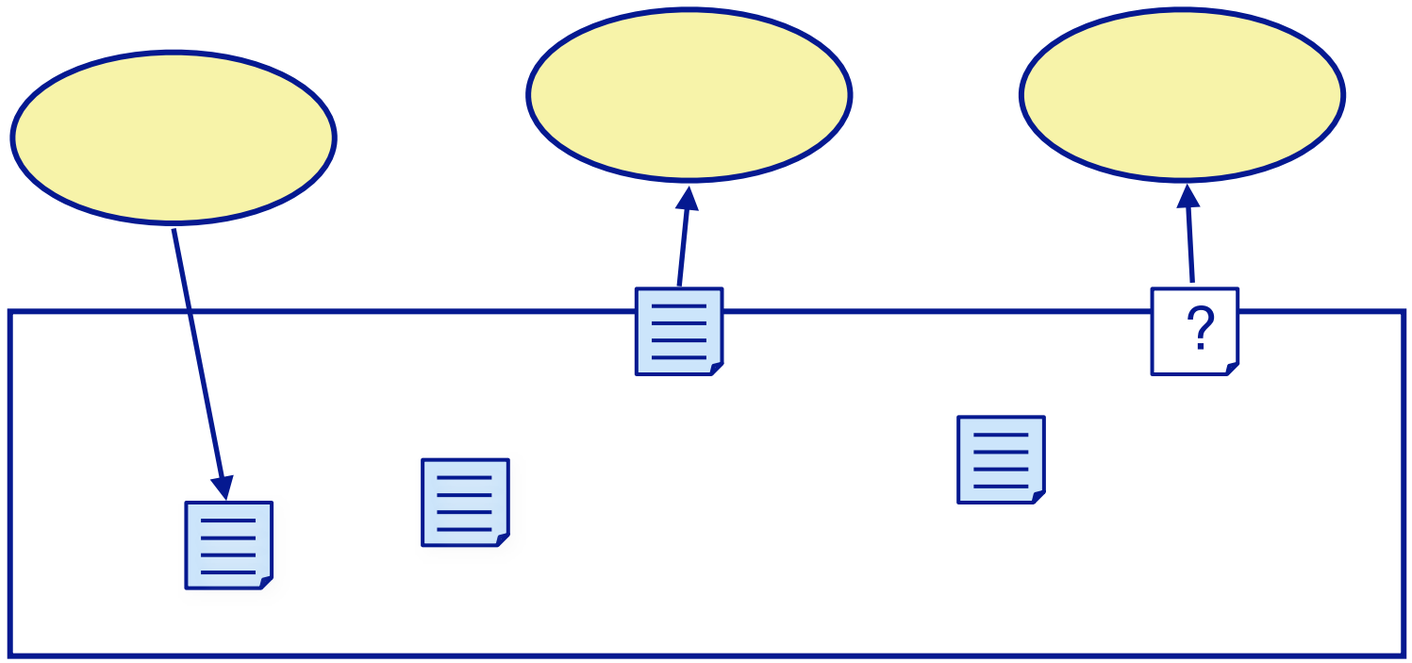
```
    public int value() { return this.value; }
    public void run() {
        int testValue = intSrc.get(); // may block
        while (testValue != 0) {
            if (testValue < this.square) {
                try {
                    new ActivePrime(testValue, lastPrime);
                } catch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

Roadmap

- What is Software Architecture?
- Three-layered application architecture
- Flow architectures
  - —Active Prime Sieve
- **Blackboard architectures**
  - —Fibonacci with Linda

Blackboard Architectures

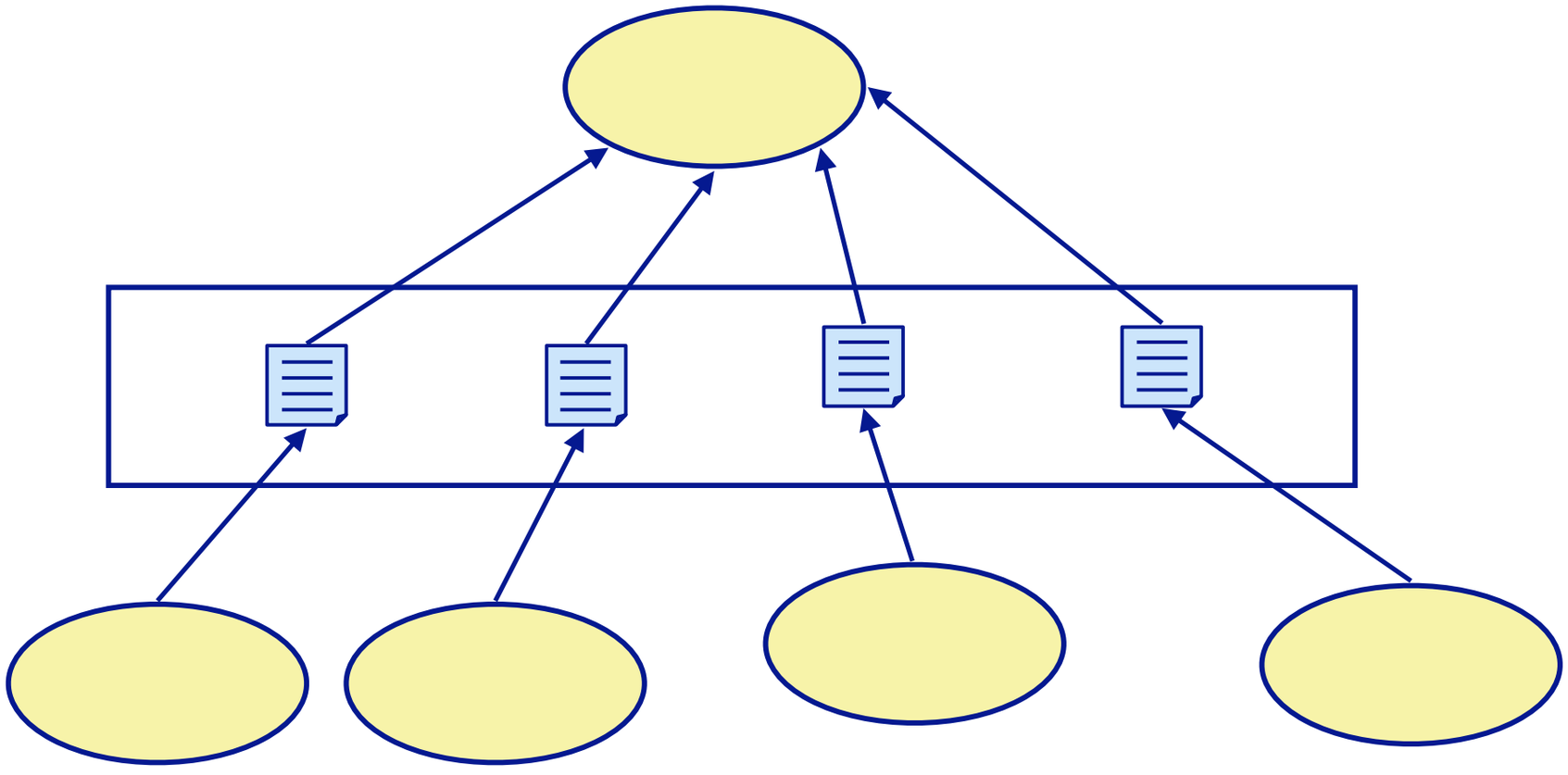
Blackboard architectures put all synchronization in a "coordination medium" where agents can exchange messages.



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).

Result Parallelism

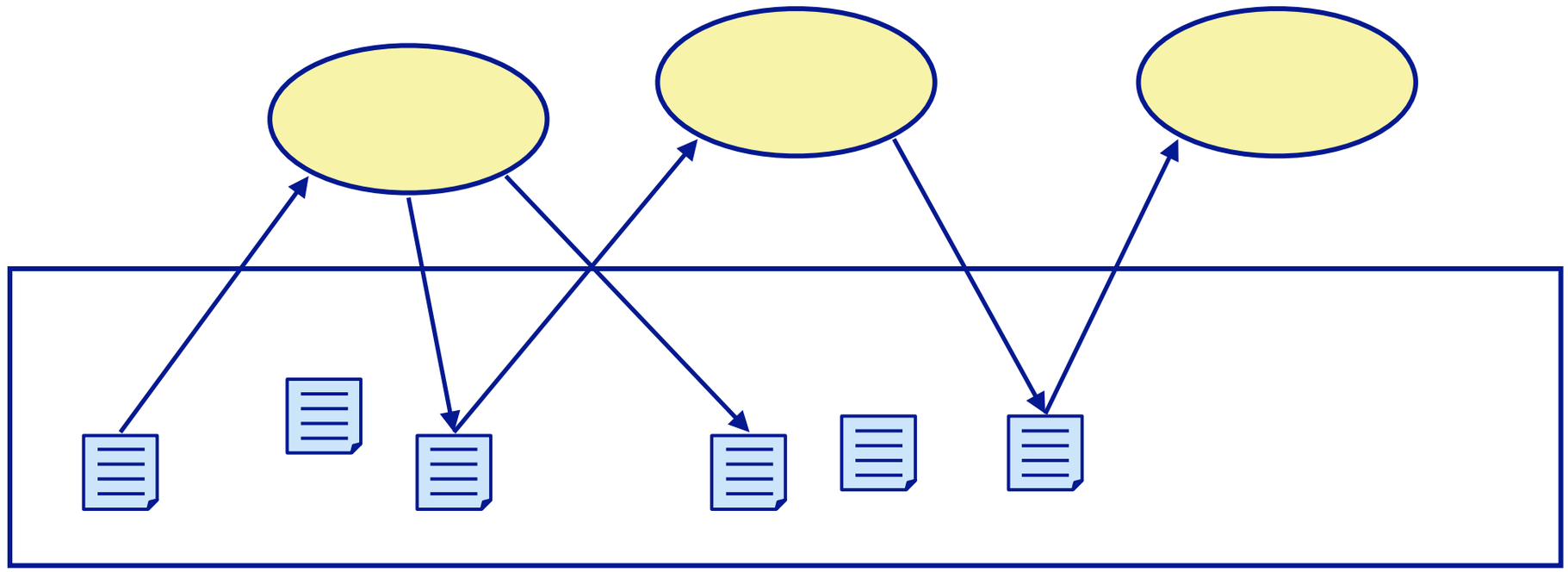
Result parallelism is a blackboard architectural style in which workers *produce parts of a more complex whole*.



Workers may be arranged hierarchically ...

Agenda Parallelism

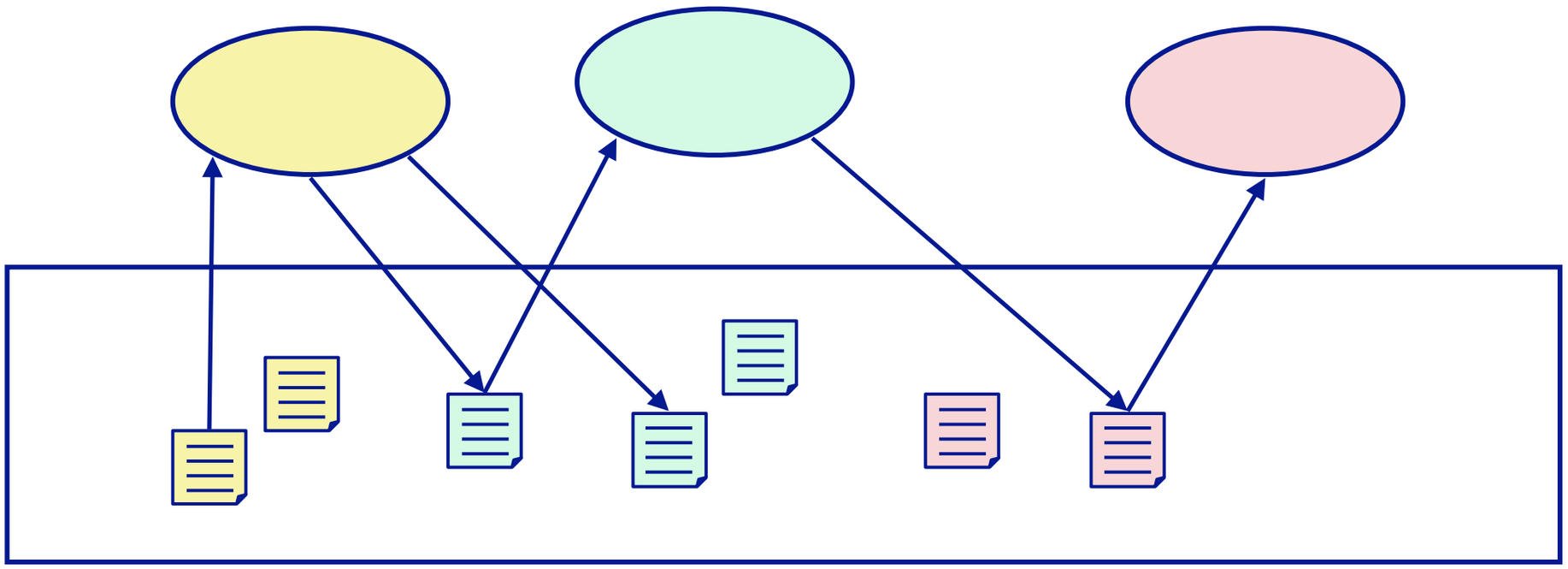
Agenda parallelism is a blackboard style in which workers *retrieve tasks to perform from a blackboard*, and may generate new tasks to perform.



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 worker is *specialized to perform a particular task*.



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

Linda

Linda is a *coordination medium*, with associated primitives for coordinating concurrent processes, that *can be added to an existing programming language.*

The coordination medium is a tuple-space, which can contain:

—*data tuples* — tuples of primitives vales (numbers, strings ...)

—*active tuples* — expressions which are evaluated and eventually turn into data tuples

In addition to the article and book by Carriero and Gelernter, see also:

https://en.wikipedia.org/wiki/Linda_(coordination_language)

Linda primitives

`out(T)`	*output* a tuple T to the medium (non-blocking) e.g., out(“employee”, “pingu”, 35000)
`in(S)`	*(destructively) input* a tuple matching S (blocking) e.g., in(“employee”, “pingu”, ?salary)
`rd(S)`	*(non-destructively) read* a tuple (blocking)
`inp(S)`	*try to input* a tuple report success or failure (non-blocking)
`rdp(S)`	*try to read* a tuple report success or failure (non-blocking)
`eval(E)`	evaluate E in a *new process* leave the result in the tuple space

The output primitive `out(T)` always succeeds, adding a tuple to the tuple space.

The input primitives `in(S)` and `rd(S)` are blocking, and only succeed if a tuple is found that matches the query pattern `S`. A query pattern is a tuple that contains both values and variables (i.e., `?salary` in the example). The primitives `inp` and `rdp` are non-blocking, but of course could lead to a busy-waiting design if used indiscriminately.

The `eval` primitive takes as an argument an expression that will evaluate to a tuple.

Roadmap

- What is Software Architecture?
- Three-layered application architecture
- Flow architectures
  - —Active Prime Sieve
- Blackboard architectures
  - —**Fibonacci with Linda**

Example: Fibonacci with JavaSpaces

```
BigInteger fib(final int n) {
    Tuple tuple;
    tuple = rdp(new Tuple("Fib", n, null));          // non-blocking
    if (tuple != null) {
        return tuple.result;
    }
    if (n<2) {
        out(new Tuple("Fib", n, BigInteger.ONE));    // non-blocking
        return BigInteger.ONE;
    }
    eval("Fib", n, new Eval("fib(" + (n-1) + ")+fib(" + (n-2) + ")") {
        public BigInteger expr() { return fib(n-1).add(fib(n-2)); }
    } );
    tuple = rd(new Tuple("Fib", n, null));           // blocking
    return tuple.result;
}   // Post-condition: rdp("Fib",n,null) != null
```

The postcondition of fib(n) is that the tuple space will hold tuples of the form (“Fib”,k,f_k), for all non-negative integer values of k up to n, and f_k is the k_{th} Fibonacci number. For n=0 or 1, the corresponding tuple is directly written, if it does not exist. For all n≥2, a process is spawned to recursively establish the postcondition for smaller values, and then compute and output the new tuple.

Note that “`null`” in a `rdp` pattern acts as a wildcard.

We are using the fly implementation of tuple spaces:

https://github.com/fly-object-space

NB: Works with Java 1.5 only.

Accessing the tuple space

```
public class Tuple {
    public String functionName;
    public Integer argument;
    public BigInteger result;
    ...
}
```

```
private Tuple rdp(Tuple template) {
    return tupleSpace.read(template, ZeroWaitTime);
}
private Tuple rd(Tuple template) {
    return tupleSpace.read(template, WaitTime);
}
private Tuple inp(Tuple template) {
    return tupleSpace.take(template, ZeroWaitTime);
}
private void out(Tuple template) {
    tupleSpace.write(template, LeaseTime);
}
private void eval(String fn, final Integer arg, final Eval eval) {
    new Thread() {
        public void run() { out(new Tuple("Fib", arg, eval.expr())); }
    }.start();
}
```

We wrap a JavaSpaces implementation to resemble Linda more closely.

NB: Print statements have been removed from this version.

```
Computing fib(5)
rdp(Tuple("Fib", 5, null)) = null
eval("Fib", 5, fib(4)+fib(3))
rd(Tuple("Fib", 5, null)) [blocks]
rdp(Tuple("Fib", 4, null)) = null
eval("Fib", 4, fib(3)+fib(2))
rd(Tuple("Fib", 4, null)) [blocks]
rdp(Tuple("Fib", 3, null)) = null
eval("Fib", 3, fib(2)+fib(1))
rd(Tuple("Fib", 3, null)) [blocks]
rdp(Tuple("Fib", 2, null)) = null
eval("Fib", 2, fib(1)+fib(0))
rd(Tuple("Fib", 2, null)) [blocks]
rdp(Tuple("Fib", 1, null)) = null
out(Tuple("Fib", 1, 1))
rdp(Tuple("Fib", 0, null)) = null
out(Tuple("Fib", 0, 1))
out(Tuple("Fib", 2, 2))
rd(Tuple("Fib", 2, 2)) [returns]
rdp(Tuple("Fib", 1, null)) = Tuple("Fib", 1, 1)
out(Tuple("Fib", 3, 3))
rd(Tuple("Fib", 3, 3)) [returns]
rdp(Tuple("Fib", 2, null)) = Tuple("Fib", 2, 2)
out(Tuple("Fib", 4, 5))
rd(Tuple("Fib", 4, 5)) [returns]
rdp(Tuple("Fib", 3, null)) = Tuple("Fib", 3, 3)
out(Tuple("Fib", 5, 8))
rd(Tuple("Fib", 5, 8)) [returns]
DONE: fib(5) = 8
```

Run this twice. The second time will be faster since all tuples have been cached. (Comment out the line to empty tuple space.)

See how very large fibonacci values can be computed.

What you should know!

- *What is a Software Architecture?*
- *What are advantages and disadvantages of Layered Architectures?*
- *What is a Flow Architecture? What are the options and tradeoffs?*
- *What are Blackboard Architectures? What are the options and tradeoffs?*
- *How does result parallelism differ from agenda parallelism?*
- *How does Linda support coordination of concurrent agents?*

Can you answer these 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?*
- *What will happen if you start two concurrent Fibonacci computations?*