13. 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


Roadmap

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

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:

> Synchronization policies of different layers may conflict
  —E.g., nested monitor lockouts

> Ground actions may need to know current policy
  —E.g., blocking vs. failing
Roadmap

> What is Software Architecture?
> Three-layered application architecture
> **Flow architectures**
  — Active Prime Sieve
> Blackboard architectures
  — Fibonacci with Linda
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
```
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).
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:
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.
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 values to test from an *IntSource*:

```java
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

```java
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();
    }
```
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 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 is a blackboard architectural style in which workers produce parts of a more complex whole.
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 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 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 values (numbers, strings ...)
— *active tuples* — expressions which are evaluated and eventually turn into data tuples
## Linda primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out(T)</td>
<td><em>output</em> a tuple T to the medium (non-blocking) e.g., out(“employee”, “pingu”, 35000)</td>
</tr>
<tr>
<td>in(S)</td>
<td><em>(destructively) input</em> a tuple matching S (blocking) e.g., in(“employee”, “pingu”, ?salary)</td>
</tr>
<tr>
<td>rd(S)</td>
<td><em>(non-destructively) read</em> a tuple (blocking)</td>
</tr>
<tr>
<td>inp(S)</td>
<td><em>try to input</em> a tuple report success or failure (non-blocking)</td>
</tr>
<tr>
<td>rdp(S)</td>
<td><em>try to read</em> a tuple report success or failure (non-blocking)</td>
</tr>
<tr>
<td>eval(E)</td>
<td>evaluate E in a <em>new process</em> leave the result in the tuple space</td>
</tr>
</tbody>
</table>
Roadmap

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

```java
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
```
JavaSpaces
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();
}

public class Tuple {
    public String functionName;
    public Integer argument;
    public BigInteger result;
    ...
}
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", 1, 1))
rdp(Tuple("Fib", 0, null)) = Tuple("Fib", 3, 3))
rd(Tuple("Fib", 3, 3)) [returns]
rdp(Tuple("Fib", 2, null)) = Tuple("Fib", 2, 2)
out(Tuple("Fib", 2, 2))
rdp(Tuple("Fib", 4, 5)) [returns]
rd(Tuple("Fib", 4, 5)) [returns]
rdp(Tuple("Fib", 3, null)) = Tuple("Fib", 3, 3)
out(Tuple("Fib", 3, 3))
rdp(Tuple("Fib", 5, 8)) [returns]
rd(Tuple("Fib", 5, 8)) [returns]
DONE: fib(5) = 8
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?
You are free:
  • to copy, distribute, display, and perform the work
  • to make derivative works
  • to make commercial use of the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

• For any reuse or distribution, you must make clear to others the license terms of this work.
• Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

[http://creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)