Concurrent Architectures

INF2140 Parallel Programming: Chapter 11

April 25, 2012

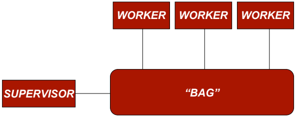
Concurrent Architectures or "Design Patterns"

- Software architectures identify software components and their interaction
    - In the context of this course architectures are process structures together with they way processes interact
- The aim is to ignore many of the details concerned with specific applications
    - Study structures that can be used in *many different situations* and applications

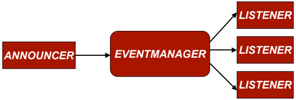
Overall


Gen Pipe Filter Pipe Filter Pipe Filter

- Architectural styles are *re-occurring patterns of components and connectors*
- We discuss three particular architectural styles:
  - Filter pipelines
  - Supervisor workers
    - Linda tuple space (for shared data)
  - Announcer listener
- Each occur commonly in concurrent and distributed systems.


WORKER WORKER WORKER

SUPERVISOR “BAG”


ANNOUNCER EVENTMANAGER

LISTENER LISTENER LISTENER

Concurrent Architectures: Filter Pipelines

Outline

- Motivation
- Concurrent Prime Sieve of Eratosthenes
- Modelling Prime Sieve in FSP
- Buffer Tolerance
- Abstraction from Filter Tasks
- Architectural Property Analysis
- (Java Prime Sieve Implementation)
- Buffering

Filter Pipelines

- Filters receive input value stream and transform them into output value stream.
- We consider filters with one input and one output stream
- Filters are connected by pipelines
  - Redirect output of one filter to input of next
  - May buffer values to de-couple processes from each other
- Example (Unix):
  - cat c340.txt 1b11.txt d50.txt | sort | less

Example: Prime Sieve

Goal: compute primes between 2 and N

Classic algorithm by Eratosthenes known as the Prime Sieve:

```
for (i:2..N) sieve[i]:=i;
for (i:2..N){
  if (sieve[i]!=0) print(i);
  for (j:i..N){
    if (sieve[j]%i=0) sieve[j]:=0;
  }
}
```

Prime Sieve

```
2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
2,3,0,5,0,7,0,9, 0,11, 0,13, 0,15, 0,17, 0,19, 0
2,3,0,5,0,7,0,0, 0,11, 0,13, 0, 0, 0,17, 0,19, 0
...
```

Prime Sieve FSP Model

- Idea:
  - Generate a Stream of numbers 2..N
  - Create one Filter for each number between 2 and N that filters all the numbers that are multiples and only outputs the others
  - Interconnect Filters by Pipes
- Leads to Filter Pipeline:

Gen

Pipe

Filter

Pipe

Filter

Pipe

Filter

PRIME Sieve in FSP (I)

```
const MAX = 5
range NUM = 2..MAX
set   S   = {[NUM],eos}

// Pipe process buffers elements from set S:
PIPE=(put[x:S]->get[x]->PIPE).

// GEN process outputs numbers from 2 to MAX
// followed by the signal eos:
GEN       = GEN[2],
GEN[x:NUM]=(out.put[x]->
            if x < MAX then GEN[x+1]
            else (out.put.eos->end->GEN)
           ).
```

Prime Sieve in FSP (II)

```
// Initialize from the first input from prev stage
FILTER=(in.get[p:NUM] -> prime[p] -> FILTER[p]
|in.get.eos -> ENDFILTER),

// Filter all inputs that are multiples of p
FILTER[p:NUM]=(in.get[x:NUM]->
                if x%p!=0then (out.put[x]->FILTER[p])
                else FILTER[p]
               |in.get.eos->ENDFILTER
               ),
// Terminate filter on eos
ENDFILTER=(out.put.eos -> end -> FILTER).
```

$$\ldots \longrightarrow_{in} FILTER[p]\ _{out} \longrightarrow \ldots$$

Prime Sieve in FSP (III)

Glue everything together:

```
||PRIMES(N=4)=
  (gen:GEN
   ||  pipe[0..N-1]:PIPE
   ||  filter[0..N-1]:FILTER)
   /{pipe[0]/gen.out,
     pipe[i:0..N-1]/filter[i].in,
     pipe[i:1..N-1]/filter[i-1].out,
     end/{filter[0..N-1].end,gen.end}}
  @{filter[0..N-1].prime,end}.
```

- *end* is made global

$$filter[i-1]_{out} \longrightarrow pipe[i] \longrightarrow_{in} filter[i]$$

Buffering

- The Prime Sieve Model so far has just one buffer slot.
    - Does it behave the same with no buffering?
    - Does it behave the same with more buffering?
    - Explosion in state space occurs if we attempt to model bigger buffer space in pipes.
- Why use buffering?
    - Performance
        - avoid context switches
        - run filters in parallel
    - Network
        - cannot avoid buffering

Unbuffered pipeline

Will our architecture work in the unbuffered case?

- Remodel and try:

Remove pipes, and directly plumb one filter into the next:

```
||PRIMESUNBUF(N=4)
  = (gen:GEN
     ||  filter[0..N-1]:FILTER)
    /{  pipe[0]/gen.out.put,
        pipe[i:0..N-1]/filter[i].in.get,
        pipe[i:1..N-1]/filter[i-1].out.put,
        end/{filter[0..N-1].end,gen.end}
     }@{filter[0..N-1].prime,end}.
```

Abstraction from Application Details

- From an architectural point of view it is not important that integers are passed as buffer elements
- We can abstract from this application detail

General Filter Pipeline

Abstract out the details of what is passed down the pipe,

- and what is actually prime:

```
||AGEN    = GEN/{out.put/out.put[NUM]}.
||AFILTER = FILTER/{out.put/out.put[NUM],
      in.get/in.get[NUM], prime/prime[NUM]}.
||APIPE  = PIPE/{put/put[NUM],get/get[NUM]}.
```

As before, but using APIPE, AGEN and AFILTER:

```
||PRIMES(N=4)=(gen:AGEN  ||  pipe[0..N-1]:APIPE
                  ||  filter[0..N-1]:AFILTER)
                  /{pipe[0]/gen.out,
                    pipe[i:0..N-1]/filter[i].in,
                    pipe[i:1..N-1]/filter[i-1].out,
                    end/{filter[0..N-1].end,gen.end}
                   }.
```

General Filter Pipeline with Buffered Pipelines

Multi-stage pipe defined using recursive definition:

```
||MPIPE(B=4) =
  if B == 1 then APIPE
  else (APIPE/{mid/get}||MPIPE(B-1)/{mid/put})
  @{put,get}.
```

As before, but using MPIPE:

```
||PRIMES(N=4)=(gen:AGEN || pipe[0..N-1]:MPIPE
              || filter[0..N-1]:AFILTER)
              /{pipe[0]/gen.out,
                pipe[i:0..N-1]/filter[i].in,
                pipe[i:1..N-1]/filter[i-1].out,
                end/{filter[0..N-1].end,gen.end}
               }.
```

```
-put-> APIPE -mid-> APIPE -mid-> APIPE -mid-> APIPE -get->
```

Architectural Property Analysis

- Refer to properties for abstract model
  - Concerned with structure and interaction
  - *not* with detailed operations
- General properties
  - Absence of deadlocks?
  - Eventual termination?
  - Ordering of results: Do filters produce results in the correct order?

Architectural Properties in FSP

- Absence of deadlocks: As usual
- Termination of the system:

```
progress TERMINATION = {end}
```

- Production of (prime) results in proper order:

```
property
PRIMEP(N=4)=PRIMEP[0],
PRIMEP[i:0..N]=
        ( when(i<N) filter[i].prime->PRIMEP[i+1]
        | end->PRIMEP).
||ORDER_CHECK=(PRIMES||PRIMEP).
```

Summary: Filter Pipelines

- Concurrent Software Architectures?
- Modelling Filters & Pipelines in FSP
- Abstraction from Filter Tasks
- Impact of Buffering
- Architectural Property Analysis
- Buffering

Next: Supervisor-Worker Architectures

Supervisor-Worker architecture model

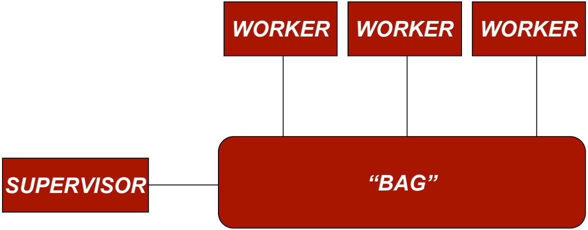
Outline

- Motivation
- Linda Tuple Spaces
- Modelling Tuple Spaces in FSP
- (Implementing Tuple Spaces in Java)
- Supervisor-Worker Model
- (Supervisor-Worker Java Implementation)

Supervisor-Worker model

Motivation

- Exploiting parallel execution on multiple processors
- Communication between different processors by a connector called **bag**
  - Supervisor creates tasks and puts them into **bag**
  - Workers pick tasks from bag and perform them
- Workers may themselves be supervisors



Linda Tuple Spaces

- is a primitive for implementing “bag” connectors.
- A **tuple** is a tagged data record:
  - Tuples are exchanged in tuple spaces using associative memory.
- Available basic operations:
  - out(“tag”,expr1,..,exprn)
  - in(“tag”,field1,..,fieldn)
  - rd(“tag”,field1,..,fieldn)
  - inp(“tag”,field1,..,fieldn)
  - rdp(“tag”,field1,..,fieldn)

Linda basic operations

- out("tag",expr1,..,exprn)

Execution completes when the expressions have been evaluated and the resulting tuple deposited in the tuple space.

- in("tag",field1,..,fieldn)

Execution blocks until the tuple space contains a tuple matching the specified fields. Input to a variable v by ?v.

- rd("tag",field1,..,fieldn)

Like in, but does not remove tuple from tuple space.

- inp("tag",field1,..,fieldn)
- rdp("tag",field1,..,fieldn)

Non-blocking versions of *in* and *rd*, returning true if there is a match.

Linda “in” operation

in("tag",field1,..,fieldn)

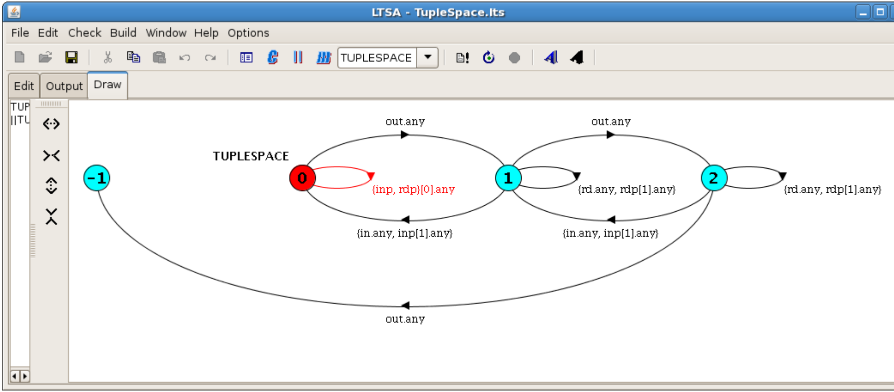
- fields are either:
  - the name of a local variable (?var) in the process calling in
  - an expression to evaluate
- A tuple in tuple-space matches the in request if:
  - the tag is identical
  - the number of fields is the same
  - the expressions equal the values in the tuple.
  - the variables have the same type as the values in the tuple.

Tuple Space Model in FSP

```
const False = 0
const True  = 1
range Bool  = False..True
const N     = 2    //Maximum Number of tuple copies
set Tuples  = {any}
set TupleAlpha = {{in,out,rd,rdp[Bool],
                   inp[Bool]}.Tuples}
TUPLE(T='any) = TUPLE[0],
TUPLE[i:0..N] = (out[T]             -> TUPLE[i+1]
      |when (i>0) in[T]             -> TUPLE[i-1]
      |when (i>0) inp[True][T]      -> TUPLE[i-1]
      |when (i==0)inp[False][T]     -> TUPLE[i]
      |when (i>0) rd[T]             -> TUPLE[i]
      |rdp[i>0][T]                  -> TUPLE[i]).

||TUPLESPACE = forall [t:Tuples] TUPLE(t).
```

Tuple Model LTS



Note: in the action names, 0 corresponds to false, 1 to true.

Tuple Space Java Implementation

```
public interface TupleSpace {
//deposits data in tuple space
public void out(String tag, Object data);
//extracts object with tag from tuple space
public Object in(String tag) throws
InterruptedException;
//reads object with tag from tuple space
public Object rd(String tag) throws
InterruptedException;
//extracts object if avail else return null
public Bool inp(String tag);
//read object if avail else return false
public Bool rdp(String tag);
}
```

Supervisor-Worker Algorithm: Outline

- Supervisor:

```
forall tasks do out("task",..)
forall results do in("result",..)
out("stop")
```

- Worker:

```
while not rdp("stop") do
  in("task",..)
  compute result
  out("result",..)
```

Supervisor-Worker Model

```
// Need a maximum on duplicate tuples:
const N = 2
// Three tuple types:
set Tuples = {task,result,stop}
// Tuple alphabet:
set TupleAlpha =
{{in,out,rd,rdp[Bool],inp[Bool]}.Tuples}
// Supervisor outputs tasks, inputs results,
// and then signals the workers to terminate:
SUPERVISOR = TASK[1],
TASK[i:1..N] = (out.task ->
               if i<N then TASK[i+1] else RESULT[1]),
RESULT[i:1..N]= (in.result ->
               if i<N then RESULT[i+1] else FINISH),
FINISH = (out.stop->end->STOP)+TupleAlpha.
```

Supervisor-Worker Model

Worker checks if it needs to stop, else inputs task, outputs results:

```
WORKER = (rdp[b:Bool].stop->
          if (!b) then (in.task->out.result->WORKER)
          else (end -> STOP) )+TupleAlpha.
```

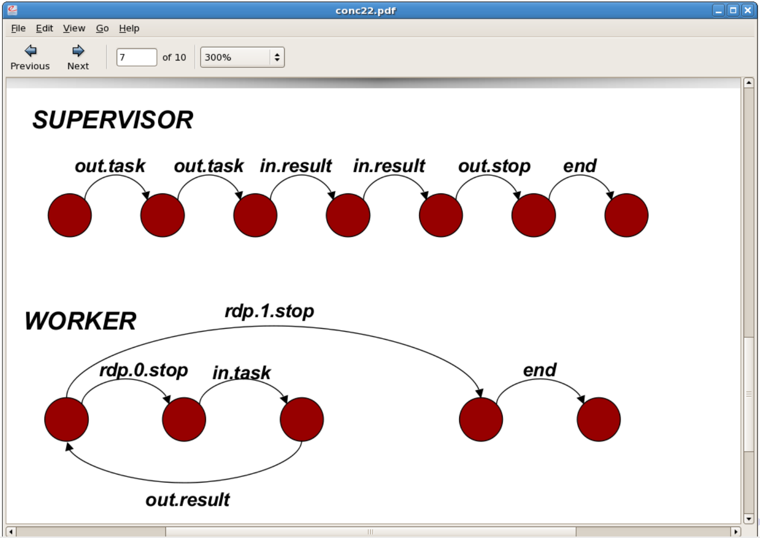
Hack to avoid spurious deadlock detection: *ended* may be repeated

```
ENDING = (end ->ENDED), ENDED = (ended->ENDED).
```

Glue it all together:

```
||SUPERVISOR_WORKER=(supervisor:SUPERVISOR
          ||{redWork,blueWork}:WORKER
          ||{supervisor,redWork,blueWork}::TUPLESPACE
          ||ENDING)
          /{end/{supervisor,redWork,blueWork}.end}.
```

Supervisor and Worker LTS



Analysis of Supervisor-Worker Model

Trace to DEADLOCK:

```
supervisor.out.task                                 1 task
supervisor.out.task                                 2 tasks
redWork.rdp.0.stop                                  no stop yet
redWork.in.task                                     1 task
redWork.out.result
supervisor.in.result
redWork.rdp.0.stop                                  no stop yet
redWork.in.task                                     0 tasks
redWork.out.result
supervisor.in.result
redWork.rdp.0.stop                                  no stop yet
supervisor.out.stop                                 stop issued
```

Supervisor only outputs stop after red worker tries to read it.

- Red is waiting for a new task that never arrives.

Deadlock Free Algorithm: Outline

- Supervisor:

```
forall tasks do out("task",..)
forall results do in("result",..)
out("task", stop)                // Note: stop as task!
```

- Worker:

```
while true do
  in("task",..)
  if value is stop               // Note: checking stop
    then out("task",stop); exit
  compute result
  out("result",..)
```

Note: stop is a special task, which can be checked by Worker.

Deadlock Free Model

```
set Tuples = {task,task.stop,result}
//Supervisor as before,except different stop method:
SUPERVISOR = TASK[1],
TASK[i:1..N] = (out.task ->
      if i<N then TASK[i+1] else RESULT[1]),
RESULT[i:1..N] = (in.result ->
      if i<N then RESULT[i+1] else FINISH),
FINISH=(out.task.stop->end->STOP)+TupleAlpha.

WORKER=(in.task -> out.result -> WORKER
      | in.task.stop->out.task.stop->end->STOP
      )+ TupleAlpha.
//Check for proper termination:
progress TERMINATION={ended}
```

Note: Worker inputs task.stop and re-emits it for other workers.

Supervisor-Worker Examples

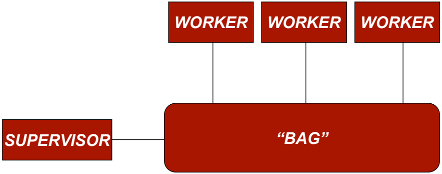
- **Booking agents**: any number of agents booking seats.
  - Clients hand out booking tasks.
  - Agents find available seats from a tuple space.
- **Computing area under a curve**
  - Approximate using rectangles
  - Parallelize task by delegating computation of different rectangles to one of 4 workers
  - Supervisor adds results computed by 4 workers
- **Concurrent execution of a sequential program**
  - find independent subproblems, and formulate tasks
  - control information flow by tuple space

Summary: Linda Tuple Spaces and Supervisor-Worker

- Motivation
- Modelling Tuple Spaces in FSP
- Implementing Tuple Spaces in Java
- Supervisor-Worker Model
- (Supervisor-Worker Java Implementation)

Note: any number of workers, and supervisors.

- need not know about each other!



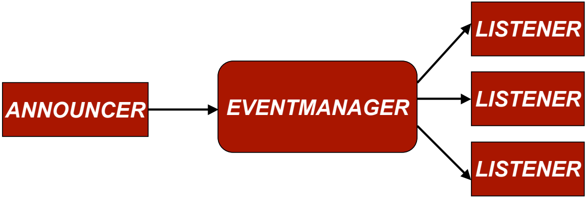
Announcer/Listener architecture model

Outline

- Motivation
- Announcer-Listener Model
- Announcer-Listener Safety and Progress
- Announcer-Listener Implementation
- Summary

Announcer/Listener architecture model: Motivation

- Notification of events:
    - Events originate in one location (announcer)
    - Communicated to multiple interested parties (listeners)
- Announcer does not know listeners.
- Listeners do not know announcer.
- Communication is managed by a connector called event manager.



Application Examples

- User Interface Frameworks:
  - AWT Listeners are ordinary objects.
  - Events are mouse clicks, button presses.
  - Events cause operations to be invoked in Listeners.
- CORBA Event Service:
  - Event Producers are Announcers
  - Event Channels are Event Managers
  - Event Consumers are Listeners
  - Used, for example in distributed stock tickers.
- PayTV broadcasting
  - customers register and pay, or deregister
  - event managers control access rights
  - announcer sends out programs
- Email (imap)

Filtering and Recursive Events

- Listeners may only be interested in a subset of events
    - They register with Event Manager using a “pattern” of events they wish to receive.
- Listeners may themselves be announcers and forward events into subsequent Event Managers.
- Listeners do not have to be active processes.

Announcer-Listener Model (part I)

```
set Listener = {a,b,c,d}
set Pattern  = {pat1,pat2}

REGISTER = IDLE,
IDLE = (register[p:Pattern] -> MATCH[p]
       | announce[Pattern]   -> IDLE),

MATCH[p:Pattern] = (announce[a:Pattern]->if(a == p)
        then(event[a] -> MATCH[p] | deregister->IDLE)
        else MATCH[p]
     | deregister -> IDLE).

||EVENTMANAGER = (Listener:REGISTER)
                 /{announce/Listener.announce}.
```

- One Event Manager for each listener.
- Here, a listener may only register for one pattern.

Announcer-Listener Model - (part II)

```
ANNOUNCER = (announce[Pattern] -> ANNOUNCER).

LISTENER(P='pattern) = (register[P] -> LISTENING),
LISTENING = ( compute  -> LISTENING
            | event[P] -> deregister -> STOP
            | event[P] -> LISTENING
            )+{register[Pattern]}.

||ANNOUNCER_LISTENER =
    (   a:LISTENER('pat1) || b:LISTENER('pat2)
     || c:LISTENER('pat1) || d:LISTENER('pat2)
     || EVENTMANAGER
     || ANNOUNCER
     || Listener:SAFE).
```

- SAFE considered next

Announcer-Listener Analysis

- Safety-Properties:
  - Listeners receive events when and only when they are registered for them
  - Listeners only receive events that match the patterns they have registered for
- Progress-Properties
  - Announcer should be able to announce events independent of state of Listeners

Example: Announcer-Listener Analysis

- Safety Properties:
  - Listener only receives events when it is registered.
  - Listener only receives the events for which it registered.

```
property SAFE = (register[p:Pattern] -> SAFE[p]),
 SAFE[p:Pattern] = ( event[p]    -> SAFE[p]
                   | deregister -> SAFE).
```

- Progress Properties:
  - The announcer can announce, no matter who is listening.

```
progress ANNOUNCE[p:Pattern] = {announce[p]}
```

Example: Announcer-Listener Model - Box Mover Game I

```
set Listener = {a,b,c,d}
set Pattern  = {pat1,pat2}

REGISTER = IDLE,
IDLE = (register[p:Pattern] -> MATCH[p]
       |announce[Pattern]   -> IDLE),
MATCH[p:Pattern] = (announce[a:Pattern]->if(a == p)
      then(event[a] -> MATCH[p]| deregister -> IDLE)
      else MATCH[p]
   | deregister -> IDLE).

||EVENTMANAGER = (Listener:REGISTER)
                 /{announce/Listener.announce}.
ANNOUNCER = (announce[Pattern] -> ANNOUNCER).

BOXMOVER(P='pattern) = (register[P]->LISTENING),
```

Example: Announcer-Listener Model - Box Mover Game II

```
LISTENING =(compute->// compute and display position
      (timeout -> LISTENING // no mouse event
      |event[P] -> timeout -> LISTENING // miss
      |event[P] -> deregister -> STOP )       // hit
  )+{register[Pattern]}.

||ANNOUNCER_LISTENER =
     ( a:BOXMOVER('pat1) || b:BOXMOVER('pat2)
    || c:BOXMOVER('pat1) || d:BOXMOVER('pat2)
    || EVENTMANAGER || ANNOUNCER || Listener:SAFE ).

progress ANNOUNCE[p:Pattern] = {announce[p]}
property SAFE    = (register[p:Pattern] -> SAFE[p]),
  SAFE[p:Pattern]= (event[p] -> SAFE[p]
                   | deregister -> SAFE).
```

Summary

- Filter pipelines
  - buffering with 0 or more slots
  - one-to-one, but may be generalized to many-to-many
- Supervisor workers and Linda tuple space
  - any-to-any (without knowing identities)
- Announcer-Listener
  - one-to-many
  - broadcasting