Distributed Process Networks in Java

Thomas M. Parks   David Roberts
Colgate University

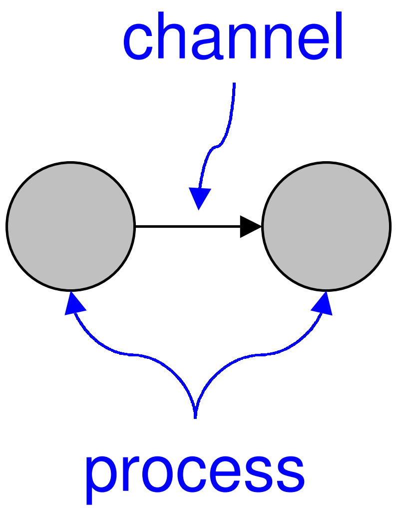
International Workshop on
Java for Parallel and Distributed Computing

Outline

- Introduction to Process Networks
- Kahn’s Formal Model
- Java Implementation of Process Networks
- Distributed Process Networks
- Conclusion

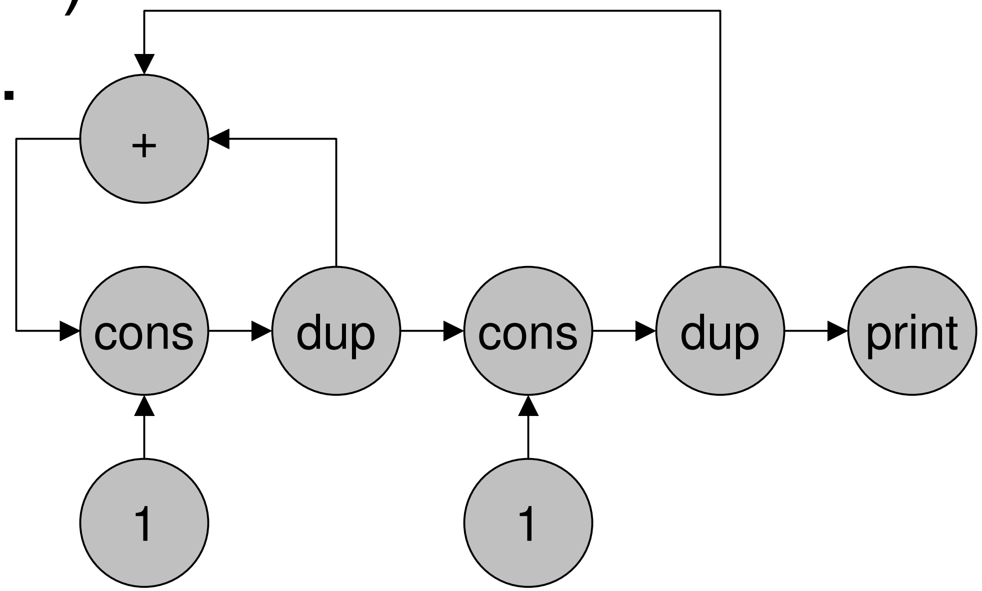
Introduction to Process Networks

- Concurrent processes
- FIFO channels, unlimited capacity
- Blocking reads
- Implementation
  - Limited channel capacity
  - Blocking writes



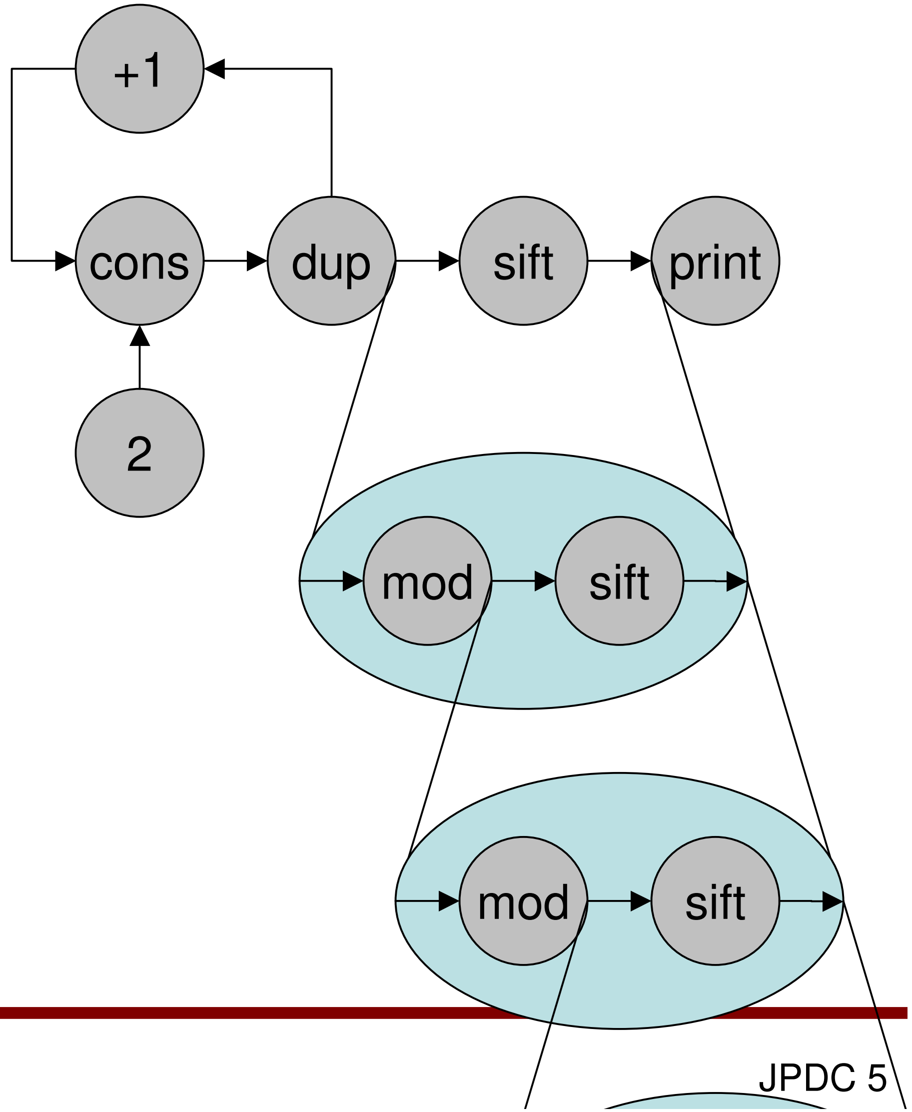
Fibonacci

- Compute Fibonacci numbers
  $F(n) = F(n-1) + F(n-2)$
  1, 1, 2, 3, 5, 8, 13…
- Directed cycles
  - Limit token production
  - Limit parallelism



Eratosthenes

- Compute prime numbers with the Sieve of Eratosthenes 2, 3, 5, 7, 11, 13, 17…
- Self-modifying program graph
- Pipelined parallelism



Outline

- Introduction to Process Networks
- Kahn’s Formal Model
- Java Implementation of Process Networks
- Distributed Process Networks
- Conclusion

Kahn’s Formal Model

- Channel → Stream $X = [x_1, x_2, x_3, \ldots]$
  - Prefix order $[3, 1, 4] \bullet [3, 1, 4, 1, 5, 9]$
- Process → Function $Y = \mathrm{f}(X)$
- Blocking reads → Monotonic functions

$$X \bullet Y \bullet \mathrm{f}(X) \bullet \mathrm{f}(Y)$$

- Iterative solution

$$X_0 = \bullet$$

$$X_{i+1} = \mathrm{f}(X_i)$$

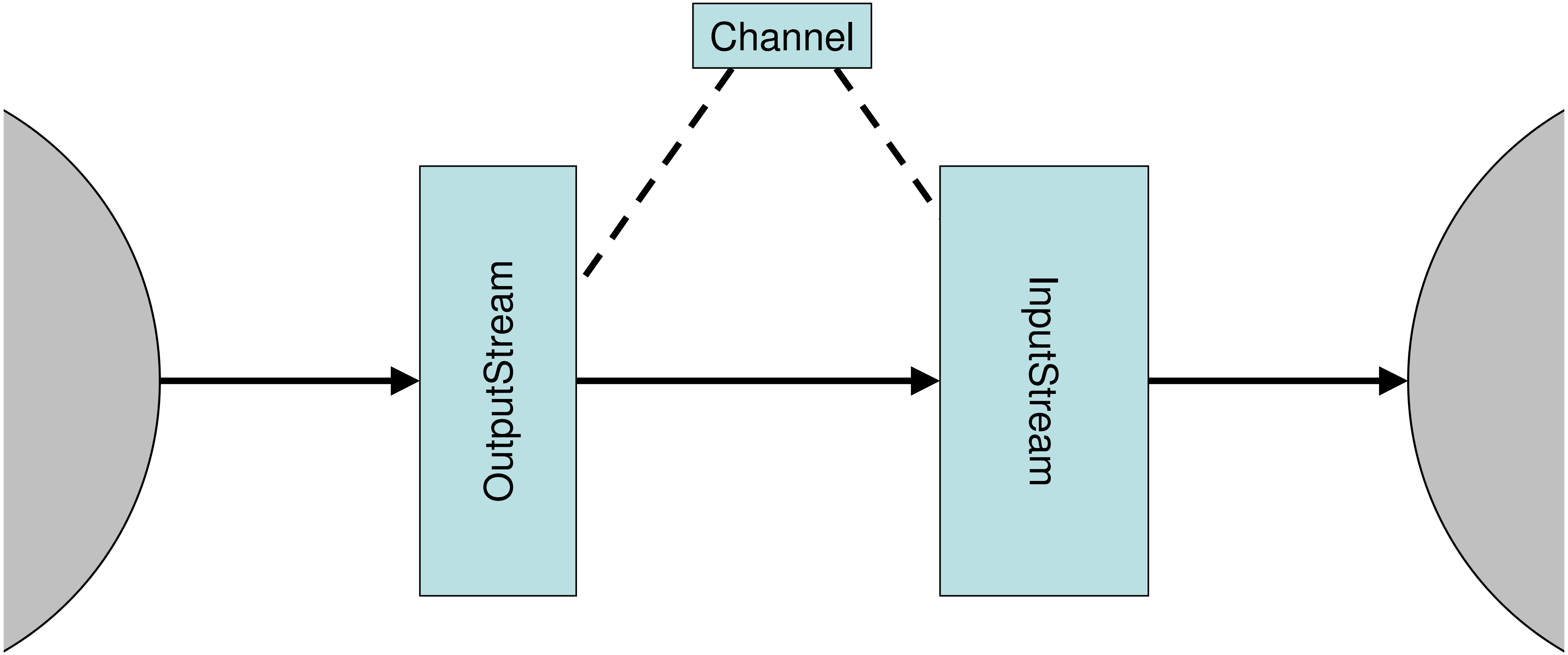
$$X_0 \bullet X_1 \bullet X_2 \bullet X_3 \bullet \ \ldots$$

→ Unique limit → Determinism

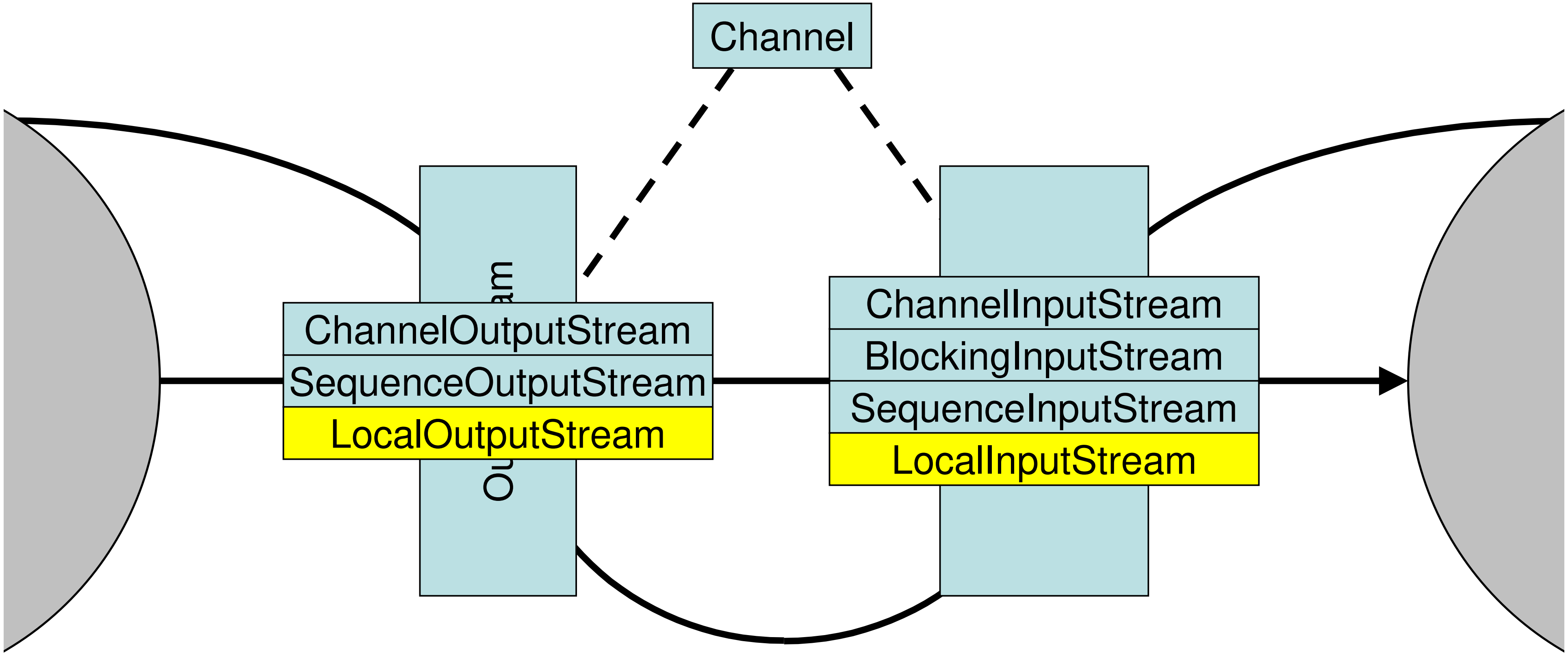
Outline

- Introduction to Process Networks
- Kahn's Formal Model
- Java Implementation of Process Networks
  – Streams
  – Processes
- Distributed Process Networks
- Conclusion

Process Networks in Java



Process Networks in Java



BlockingInputStream

- `read` never returns early
- `skip` never returns early
- `mark`, `reset` not allowed
- `available` always returns 0
- Overcomes "limitations" of Java API to provide blocking reads required for determinism

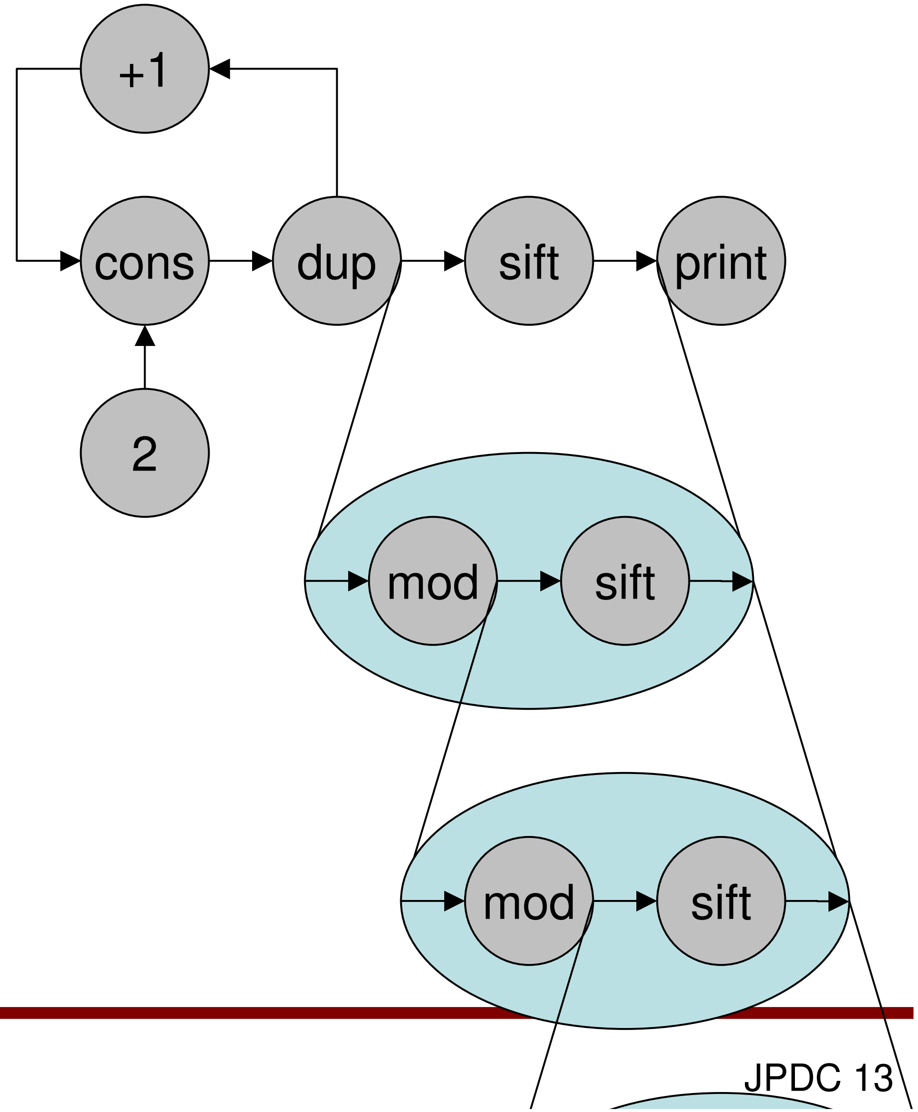
ChannelInputStream
BlockingInputStream
SequenceInputStream
LocalInputStream

IterativeProcess

```
public void run()
{   try
    {   onStart();
        if(iterations > 0)
            while(iterations-- > 0)
                step();
        else
            while(true)
                step();
    }
    catch(IOException ignored) {}
    finally { onStop(); }
}
```

Exception Propagation

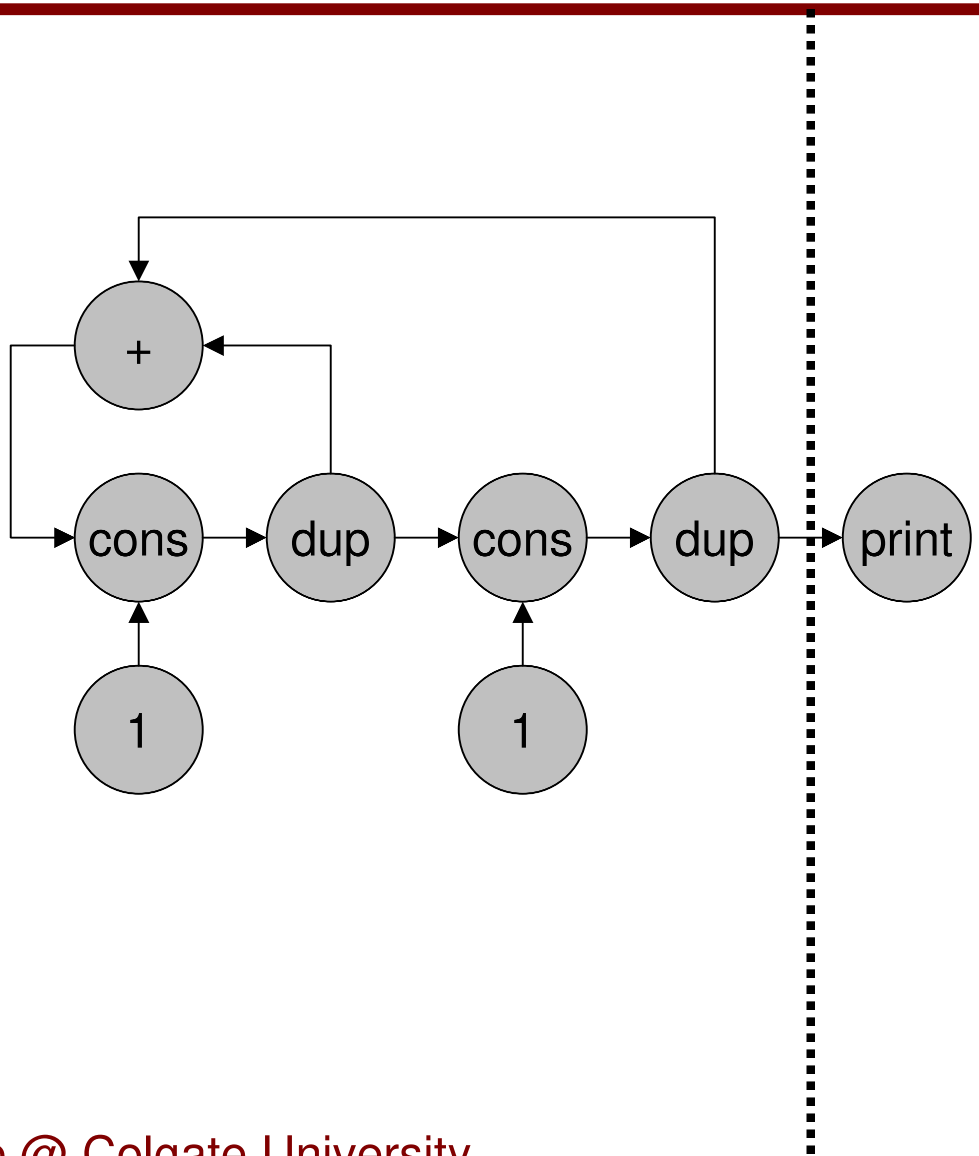
- Compute all primes less than N
- Compute the first N primes



Outline

- Introduction to Process Networks
- Kahn's Formal Model
- Java Implementation of Process Networks
- Distributed Process Networks
- Conclusion

Distributed Process Networks

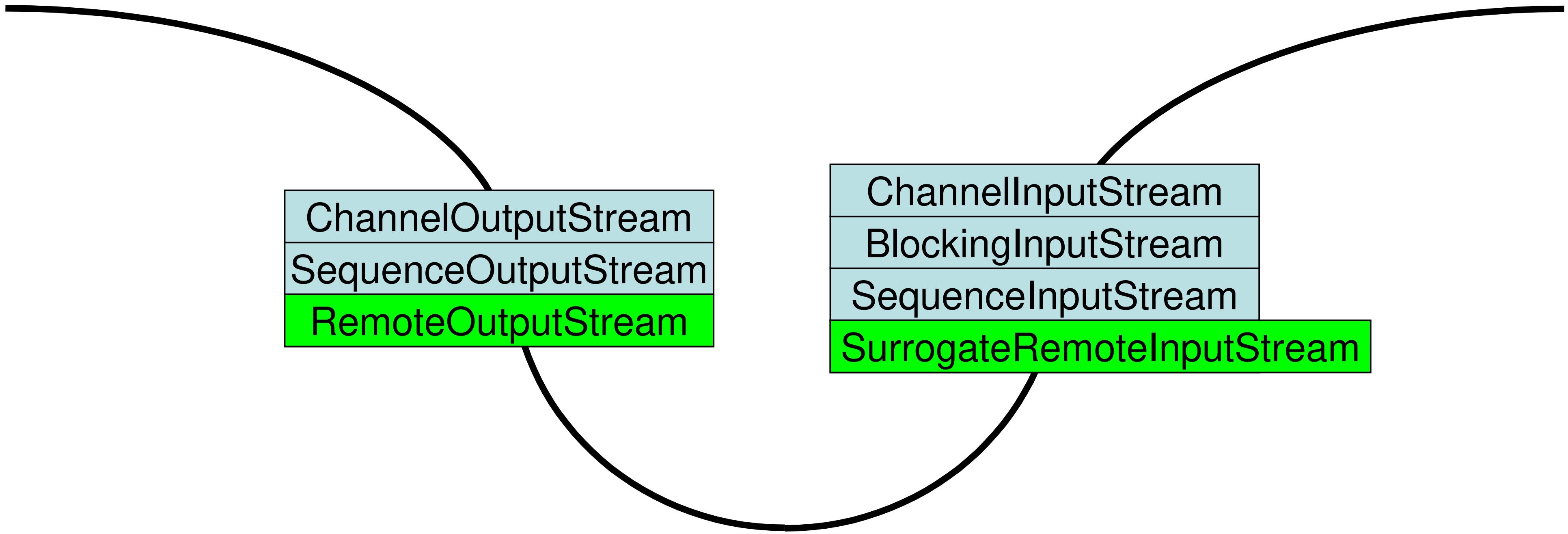


ComputeServer

```
public void run()
{   ServerSocket server = new ServerSocket(PORT);
    Socket client = server.accept();
    InputStream i = client.getInputStream();
    ObjectInputStream o
        = new ObjectInputStream(i);
    while(true)
    {   Object object = o.readObject();
        if(object instanceof pn.proc.Process)
            new Thread((Runnable)object).start();
    }
}
```

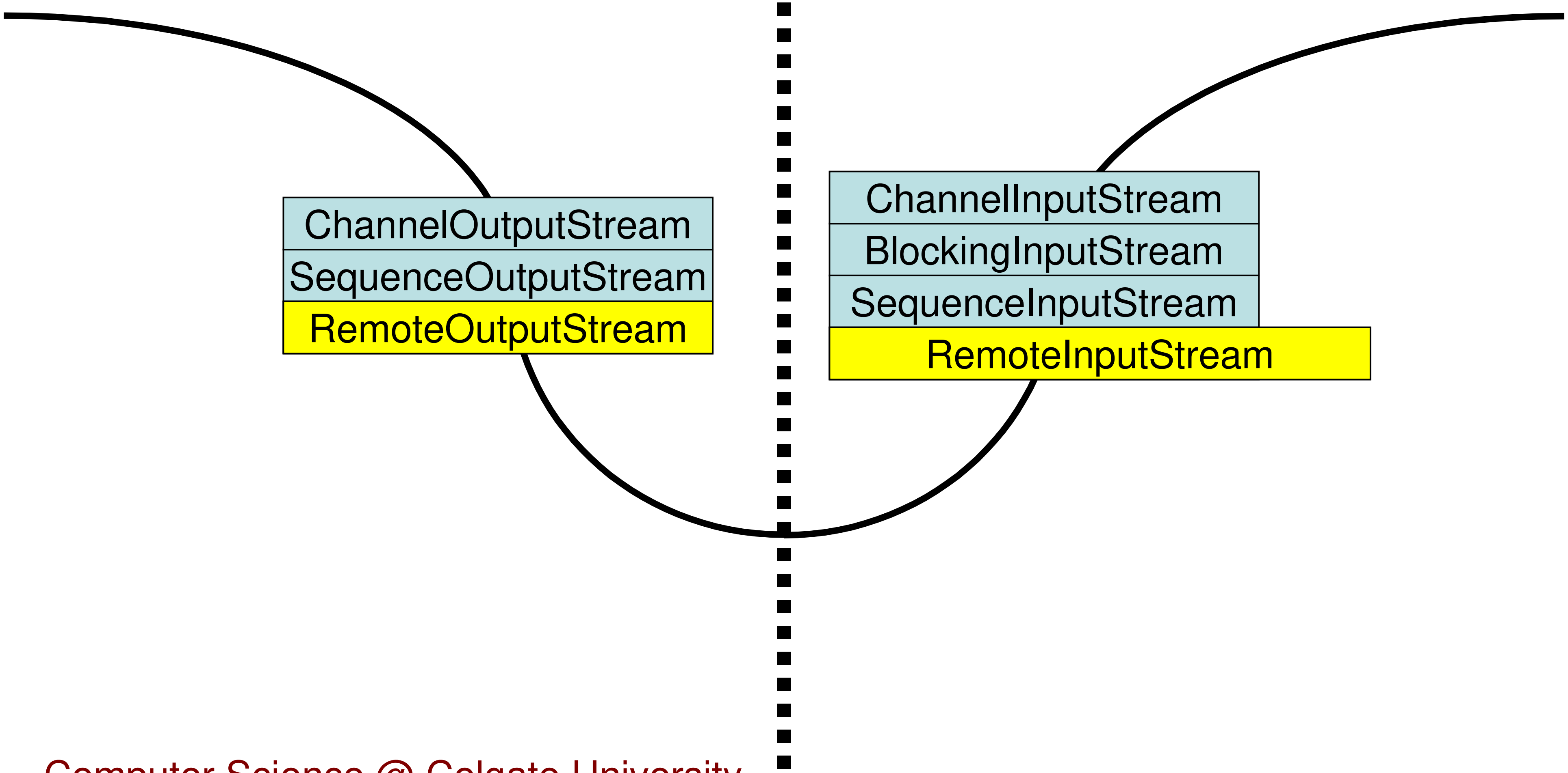
Serialization of InputStream

- Server A

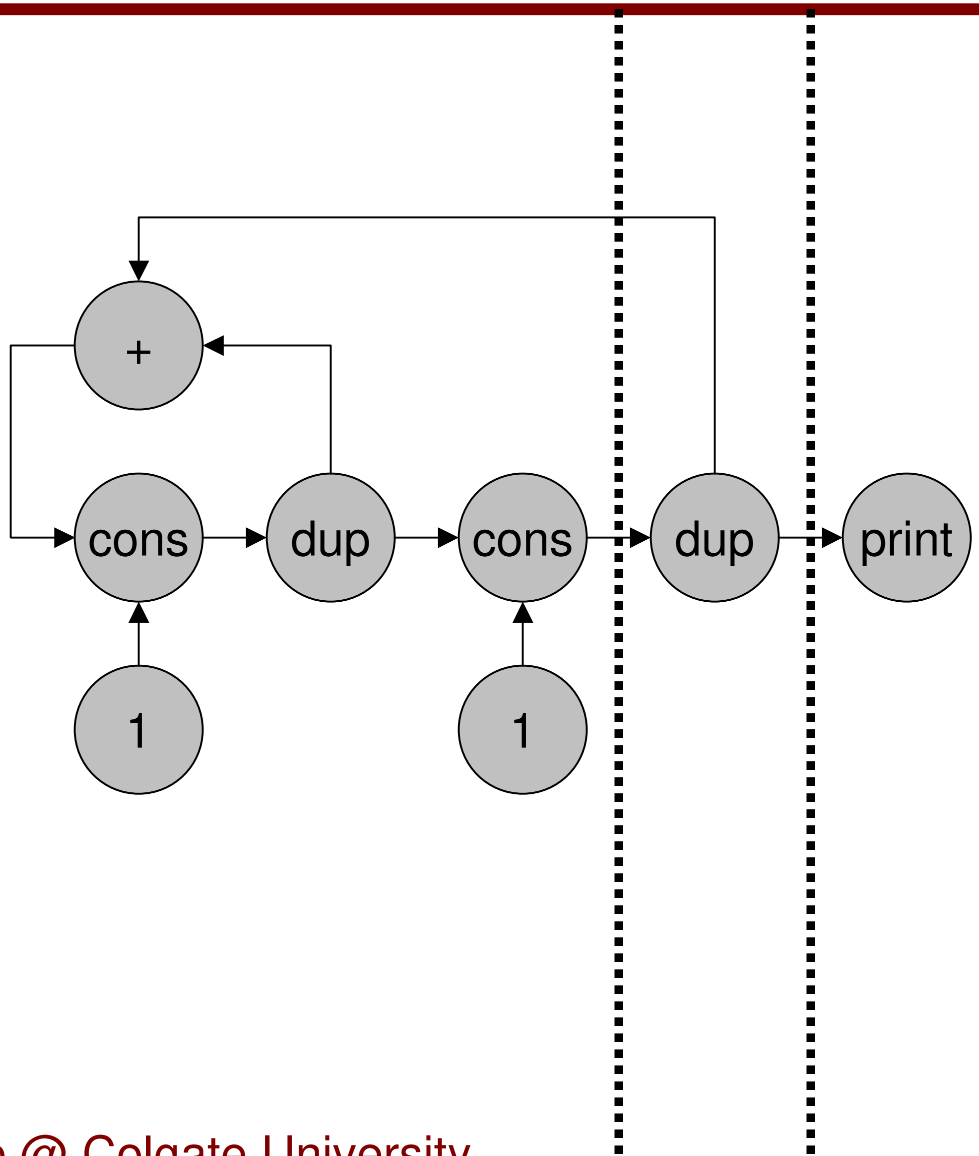


Deserialization of InputStream

- Server A
- Server B

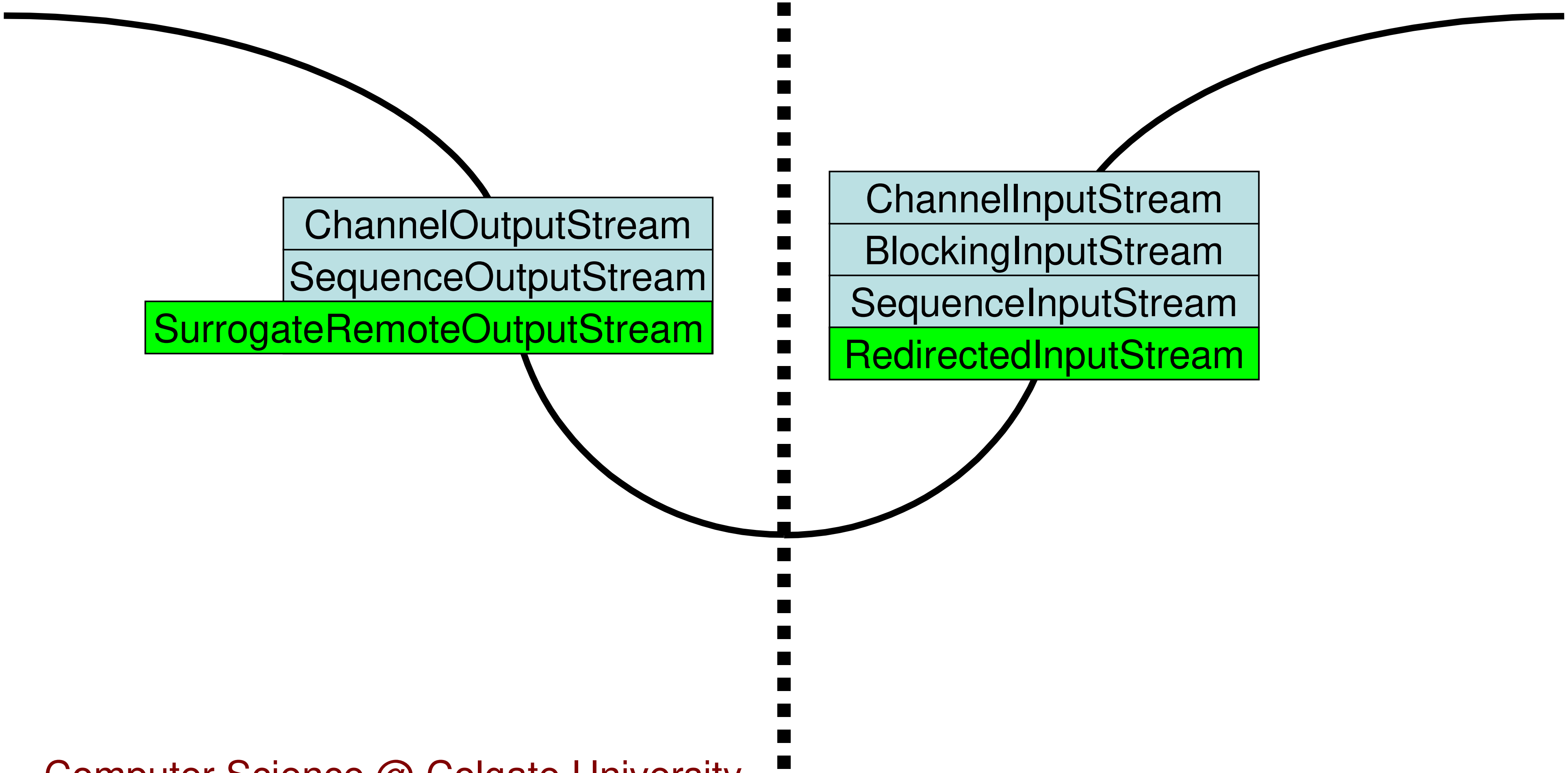


Distributed Process Networks



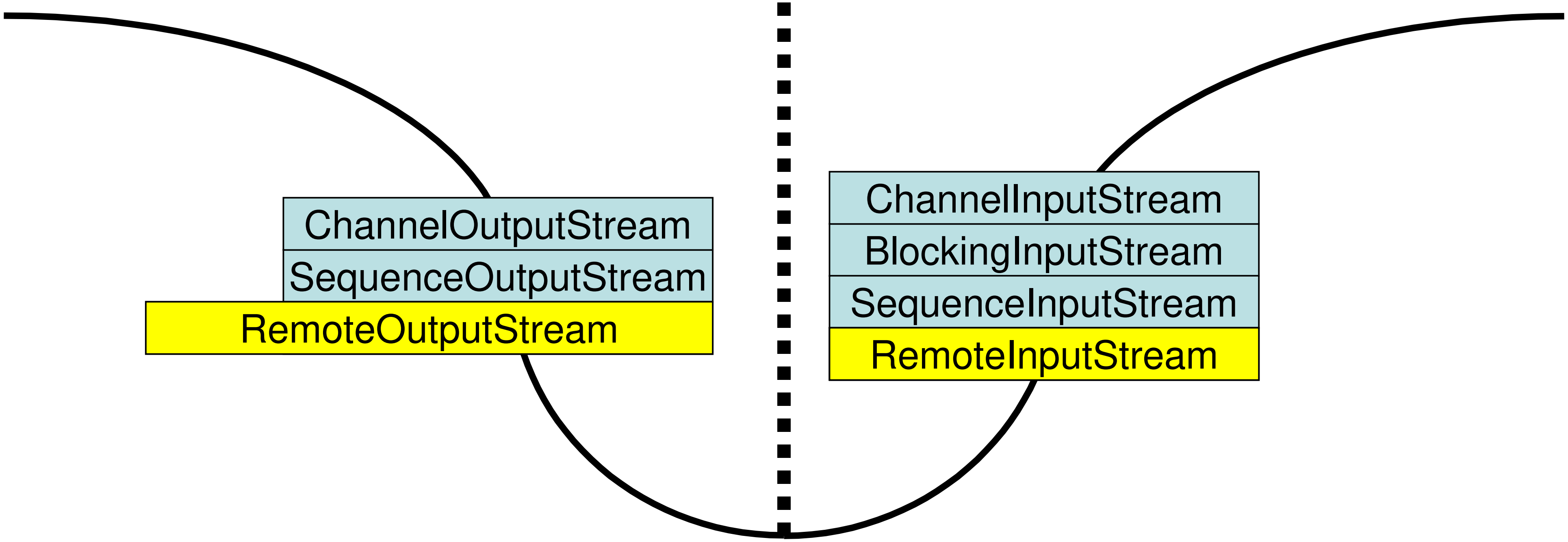
Serialization of OutputStream

- Server A
- Server B

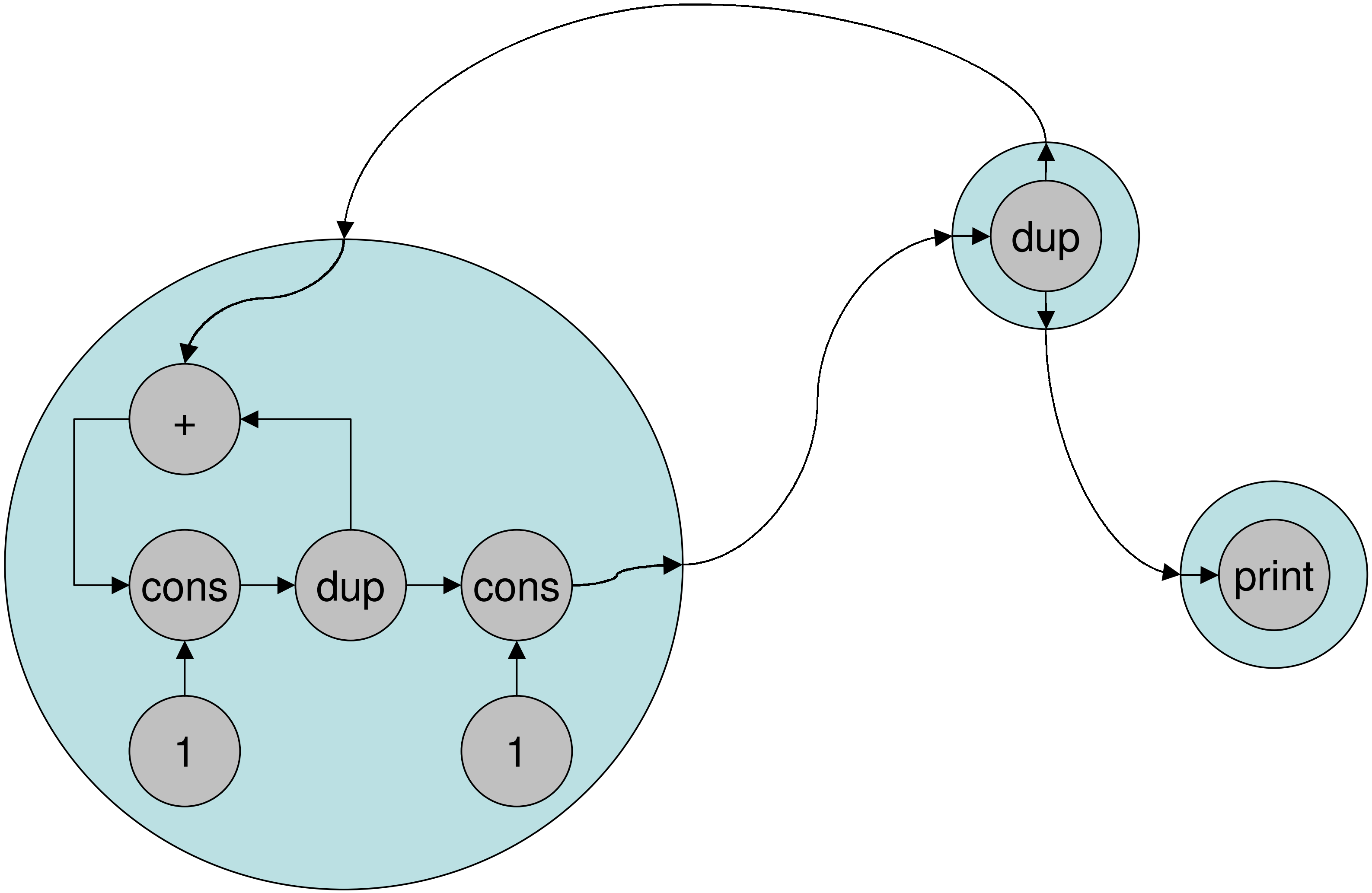


Deserialization of OutputStream

- Server C
- Server B



Direct Network Connections



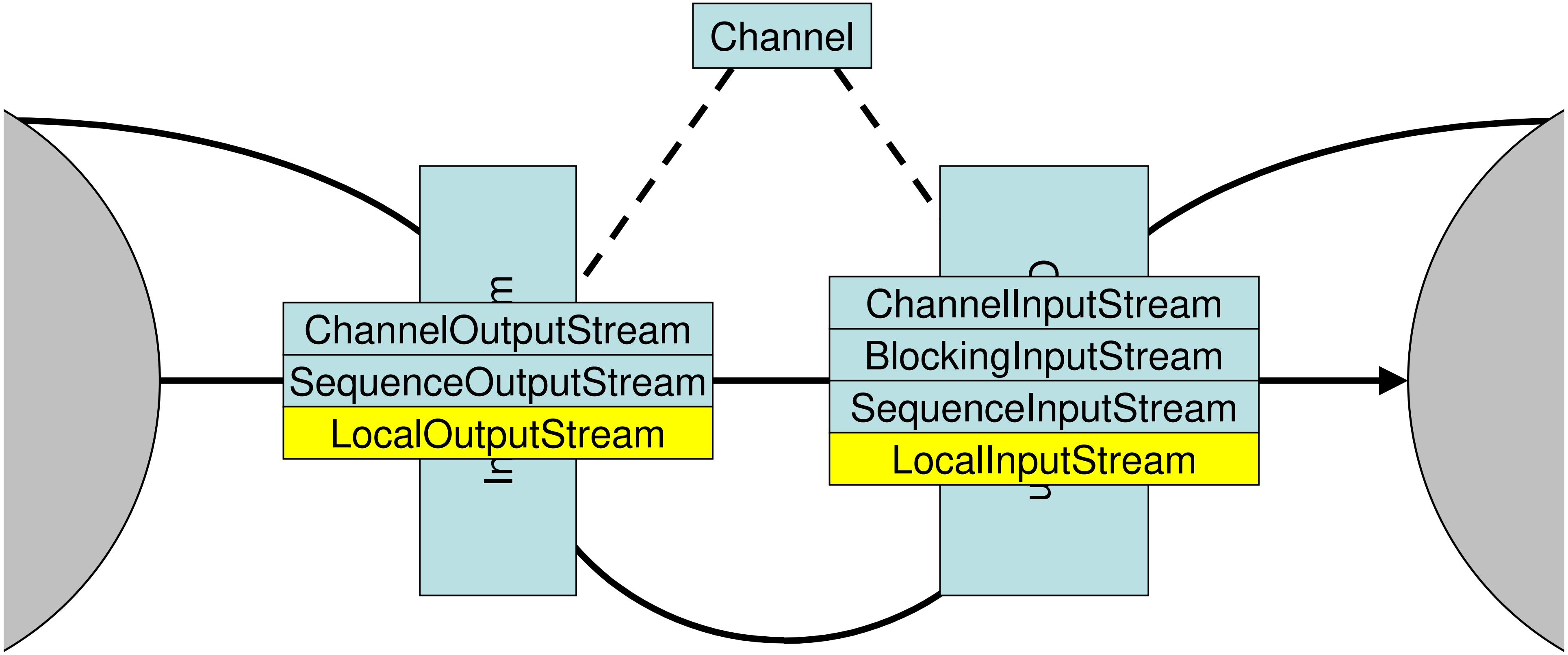
Conclusion

- Network connections established automatically during object serialization
- Graceful termination through exception propagation (even across network)
- Java implementation of process networks
  - Uniprocessor
  - Multiprocessor
  - Cluster
  - Internet

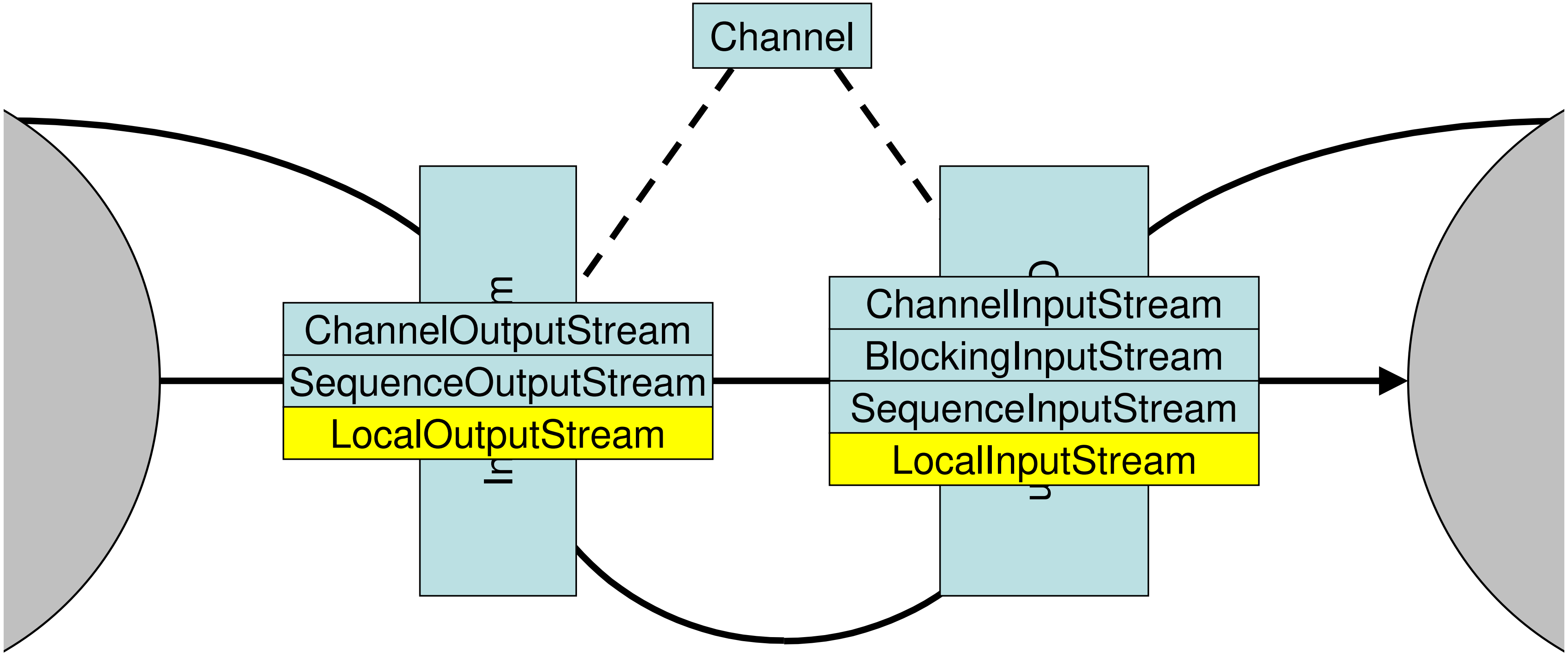
Future Work

- Process migration
- Load balancing
- Buffer management
- Distributed deadlock detection
- Web distribution of Java bytecode
- Fault tolerance
- Larger applications
- Coming soon: download from http://cs.colgate.edu/~parks

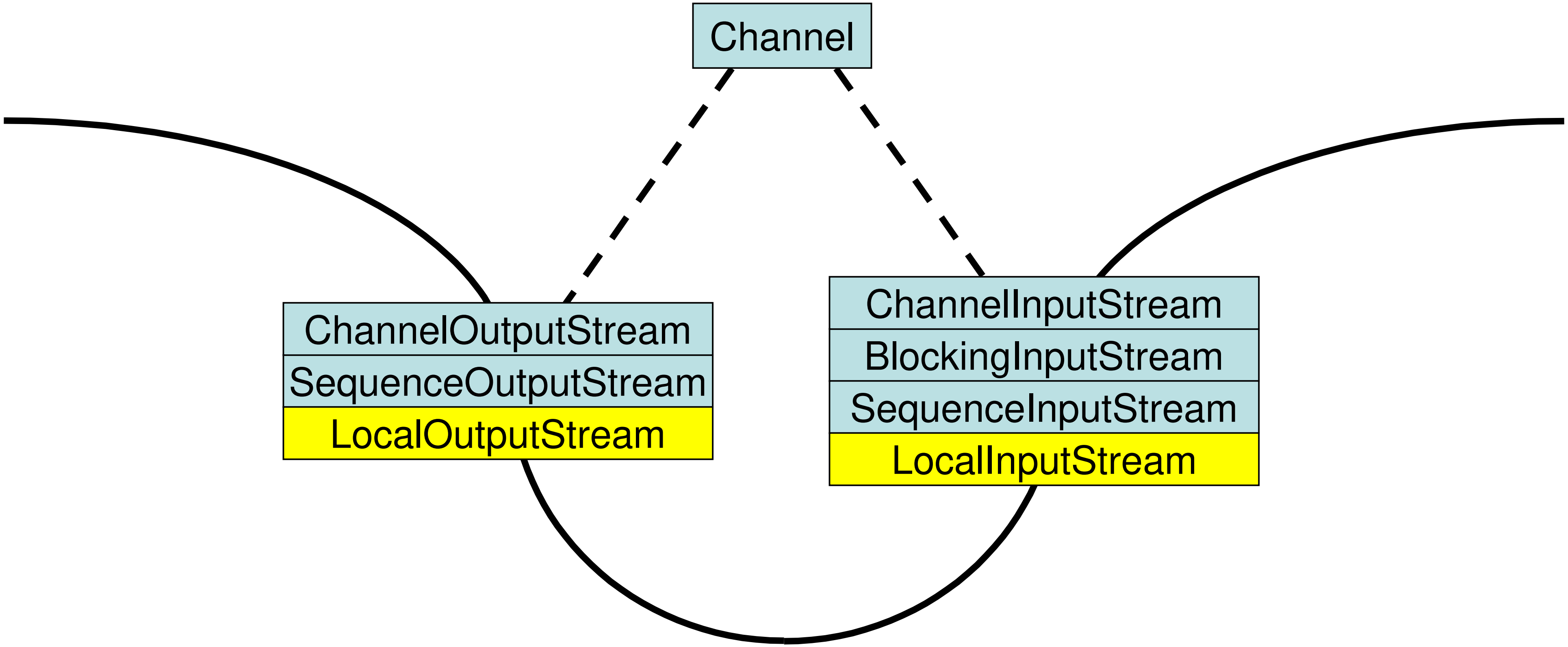
Process Networks in Java



Process Networks in Java



Process Networks in Java



Fibonacci

```
Channel ab, be, cd, df, ed, eg, fg, fh, gb;
ab = new Channel(); be = new Channel(); cd = new Channel();
df = new Channel(); ed = new Channel(); eg = new Channel();
fg = new Channel(); fh = new Channel(); gb = new Channel();

CompositeProcess p1, p2, p3;
p1 = new CompositeProcess(); p2 = new CompositeProcess();
p3 = new CompositeProcess();

p1.add(new IntSource(ab.getOutputStream(), 1));
p1.add(new IntSource(cd.getOutputStream(), 1));
p1.add(new Cons(ab.getInputStream(), gb.getInputStream(), be.getOutputStream()));
p1.add(new Cons(cd.getInputStream(), ed.getInputStream(), df.getOutputStream()));
p1.add(new Dup(be.getInputStream(), ed.getOutputStream(), eg.getOutputStream()));
p1.add(new Add(eg.getInputStream(), fg.getInputStream(), gb.getOutputStream()));

p2.add(new Dup(df.getInputStream(), fh.getOutputStream(), fg.getOutputStream()));

p3.add(new Print(90, fh.getInputStream()));

Socket s2 = new Socket("hamilton", ComputeServer.PORT);
Socket s3 = new Socket("madison", ComputeServer.PORT);
new ObjectOutputStream(s2.getOutputStream()).writeObject(p2);
new ObjectOutputStream(s3.getOutputStream()).writeObject(p3);

new Thread(p1).start();
```

Kahn's Formal Model

- Channel → Stream
- Process → Function
- Blocking reads → Continuous functions
  → Unique least fixed point
  → Determinism

Streams

- Stream $X = [x_1, x_2, x_3, \ldots]$
- Empty stream $\bullet = [\ ]$
- Prefix order $\bullet$

  $[3, 1, 4] \bullet [3, 1, 4, 1, 5, 9]$
- Tuple of streams $\mathcal{X} = (X_1, X_2, X_3, \ldots)$
- Increasing chain $X_1 \bullet X_2 \bullet X_3 \bullet \ldots$
- Least upper bound $\bullet \mathcal{X} = lim_{i \to \infty} X_i$
- Complete partial order

Processes

- Continuous $\quad f(lim_{i\rightarrow\infty} X_i) = lim_{i\rightarrow\infty} f(X_i)$
- Composition $\quad f(g(lim_{i\rightarrow\infty} X_i)) = lim_{i\rightarrow\infty} f(g(X_i))$
- Monotonic $\quad X \bullet Y \bullet f(X) \bullet f(Y)$
- Fixed point equation $\quad \mathcal{X} = f(\mathcal{X})$
- Iterative solution

$$\mathcal{X}_0 = \bullet$$

$$\mathcal{X}_{i+1} = f(\mathcal{X}_i)$$

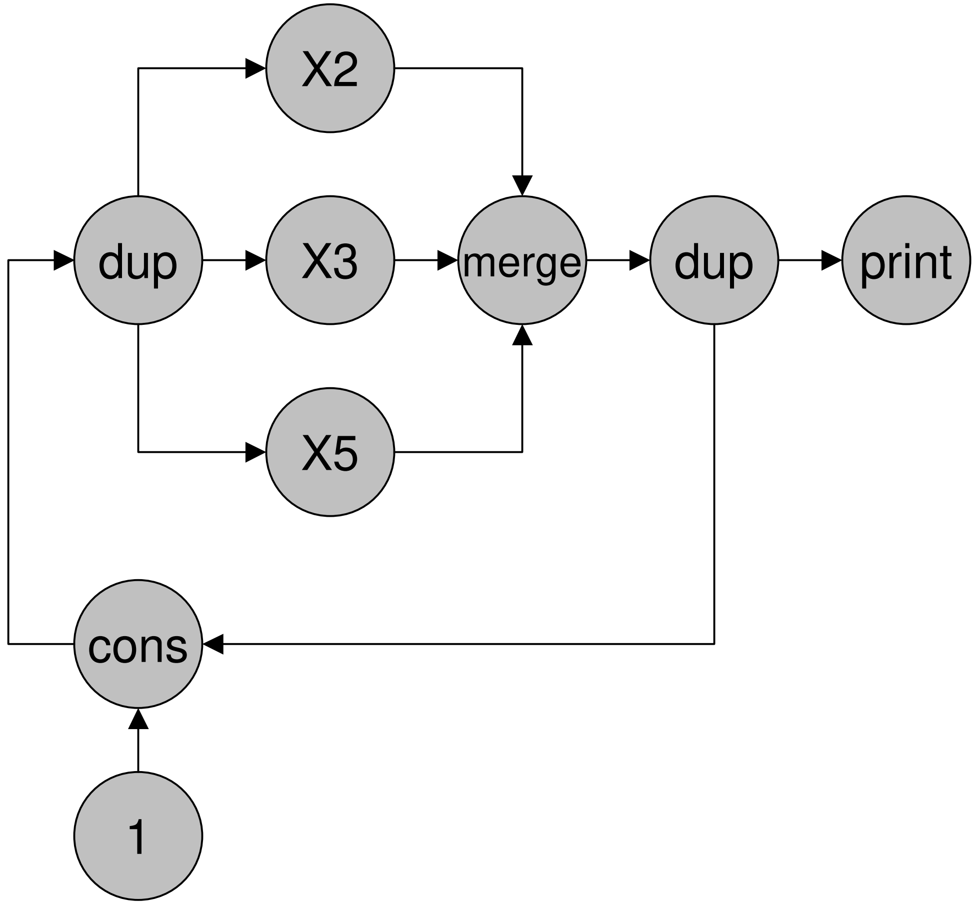
$$\mathcal{X}_0 \bullet \mathcal{X}_1 \bullet \mathcal{X}_2 \bullet \mathcal{X}_3 \bullet \ \ldots$$

Merge

```
int a = inputs[0].readInt();
int b = inputs[1].readInt();
while (true)
{   if (a < b)
    {   output.writeInt(a)
        a = inputs[0].readInt();
    }
    else if (b < a)
    {   output.writeInt(b)
        b = inputs[1].readInt();
    }
    else // discard duplicate
    {   output.writeInt(a)
        a = inputs[0].readInt();
        b = inputs[1].readInt();
    }
}
```

Dijkstra

- Compute multiples of 2, 3, 5 in order 2, 3, 4, 5, 6, 8, 9…
- Directed cycles
- Implementation
  - Limited channel capacity
  - Blocking writes
  - Deadlock (eventually)



Merge

- No directed cycles
- Implementation
  - Limited channel capacity
  - Blocking writes
  - Deadlock if capacity < N-2