Web-Scale Data Management Systems

Prof. **Oliver** Kennedy

http://www.cse.buffalo.edu/~okennedy/courses/cse704fa2012.html

okennedy@buffalo.edu

About Me

- New to UB (super excited to be here)
- 4+ years working in databases
    - And 3+ years in distributed systems before that
- Worked with analytics teams at Microsoft Azure

Office Hours

- Davis Hall 338H
  - Monday: 2:00-4:00 PM
  - Thursday: 2:00-4:00 PM
  - or by appointment

Course Goals

- Familiarize you with systems for storing and analyzing really really big data
  - Real world systems (if you want to go into industry)
  - Interesting problems (if you want to go into academia)
- Past, present, and future systems
- Give (some of) you experience working with these systems.

Course Requirements

(Everyone)

- Submit paper overviews **Each Week**
  - **1-2 paragraphs** per paper
    - What's the message of the paper?
    - What are the pros and cons of the paper's approach?
- Everyone gets 2 weeks of missed overviews

Course Requirements

(2+ Credits)

- Present 2 papers
  - 30-40 minute presentation
  - 20-30 minutes of discussion
  - Arrange to go over your presentation with me by the prior Thursday.

Course Requirements

(3+ Credits)

- A simple implementation project
  - Hadoop-based Join Algorithms
    - A Ring DHT
    - Standalone Distributed Join Algorithms
- Or develop your own project!

Course Requirements

(3+ Credits; Continued)

- Available Resources for Projects
  - A 12 core development testbed
  - $100 of Amazon Cloud Credit
    - **Be careful with this credit!**
    - **Avoid running out early!**

Course Requirements

(3+ Credits; Continued)

- Project Requirements
  - A 1-page milestone report on Nov 1
  - A short (~4 page) final report
  - A short (10-15 minute) presentation
- Confirm your project with me by Oct. 1

Dryad	Data-Flow Computation
Map Reduce & HDFS	Map/Reduce
Hive & HadoopDB	SQL on M/R
Pig & Dremel	Other Languages on M/R
MonetDB & DataCyclotron	Column Stores
Cassandra & BigTable/HBase	Semistructured Databases
Zookeeper & Percolator	Distributed Consistency
Chord & Dynamo	Distributed Hash Tables
PIQL	Enforced Scalability
Lipstick	Workflow Provenance
Borealis & DBToaster	Stream Processors

Dryad	Data-Flow Computation
Map Reduce & HDFS	Map/Reduce
Hive & HadoopDB	SQL on M/R
Pig & Dremel	Other Languages on M/R
MonetDB & DataCyclotron	Column Stores
Cassandra & BigTable/HBase	Semistructured Databases
Zookeeper & Percolator	Distributed Consistency
Chord & Dynamo	Distributed Hash Tables
Very recent (presented last month) — PIQL	Enforced Scalability
Very recent (presented last month) — Lipstick	Workflow Provenance
Borealis & DBToaster	Stream Processors

Questions about the Course?

Dryad

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly
Microsoft Research: Silicon Valley

(presented by Oliver Kennedy)

Parallel Programming

Parallel Programming

Parallel Programming

Hard Problem!

Parallel Programming

Hard Problem!

Locks!

Deadlock!

IPC!

Error Handling!

What Has Worked?

What Has Worked?

What Has Worked?

The **programmer** knows what's **data-parallel**.

The **computer** understands the **infrastructure**.

What Has Worked?

```
cat users.dat
```

What Has Worked?

```
cat users.dat
      | sed 's/\([^ ]*\).*/\1/'
      | grep 'buffalo'
      | wc -l
```

Unix pipes allow programmers to compose 'nuggets' of computation

Graph Programming

Use the same metaphor!

Join nuggets of code into a graph

Graph Programming

How about an Example?

Graph Programming

```
SELECT distinct p.objID
FROM    photoObjAll p
   JOIN neighbors n
     ON p.objID = n.objID
    AND n.objID < n.neighborObjID
   JOIN photoObjAll l
     ON l.objID = n.neighborObjID
    AND SimilarColor(l.rgb,n.rgb)
```

Graph Programming

```
SELECT distinct p.objID
FROM    photoObjAll p
  JOIN neighbors n
    ON p.objID = n.objID
   AND n.objID < n.neighborObjID
  JOIN photoObjAll l
    ON l.objID = n.neighborObjID
   AND SimilarColor(l.rgb,n.rgb)
```

Join X

Graph Programming

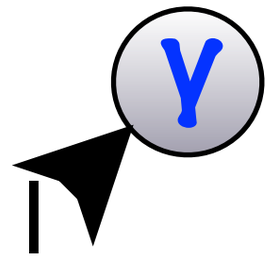
```
SELECT distinct p.objID
FROM   photoObjAll p
  JOIN neighbors n
    ON p.objID = n.objID
   AND n.objID < n.neighborObjID
  JOIN photoObjAll l
    ON l.objID = n.neighborObjID
   AND SimilarColor(l.rgb,n.rgb)
```

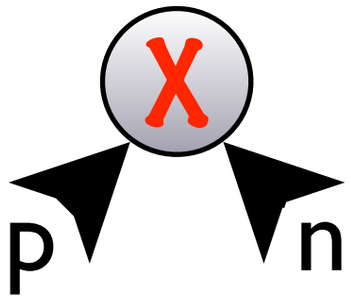
Join X

Join Y

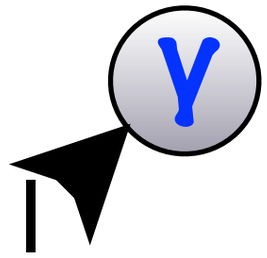
Graph Programming

Graph Programming



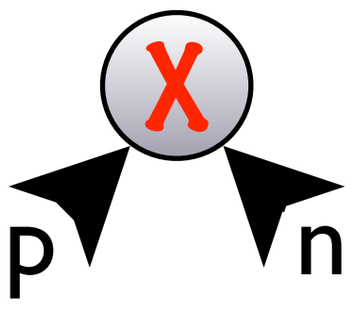


Graph Programming

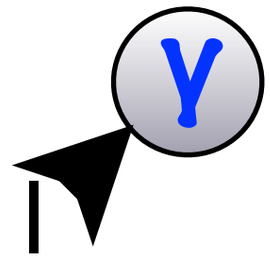


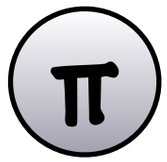
Read p⋈n

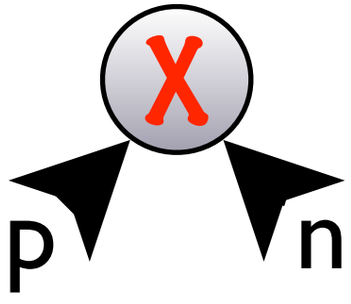
Emit n.neighborID : p⋈n



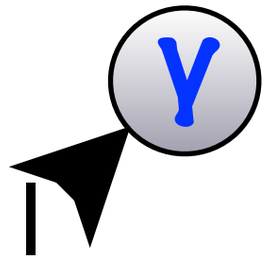
Graph Programming

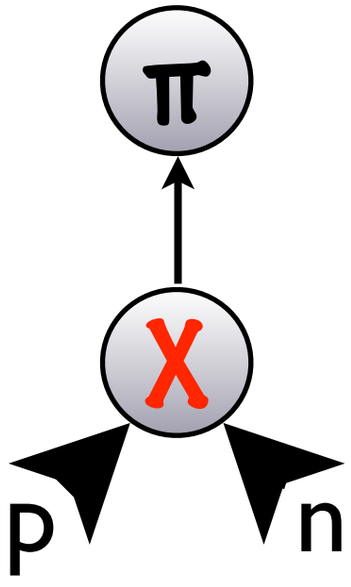




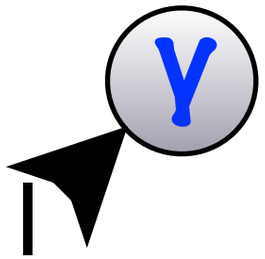


Graph Programming

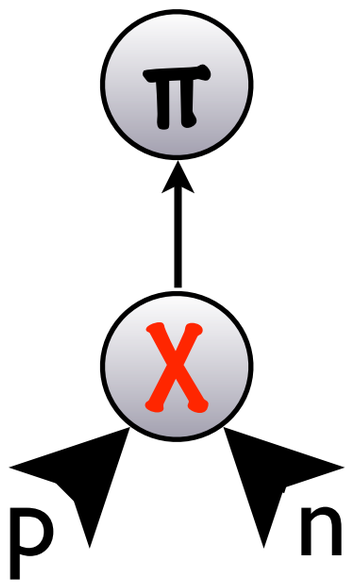




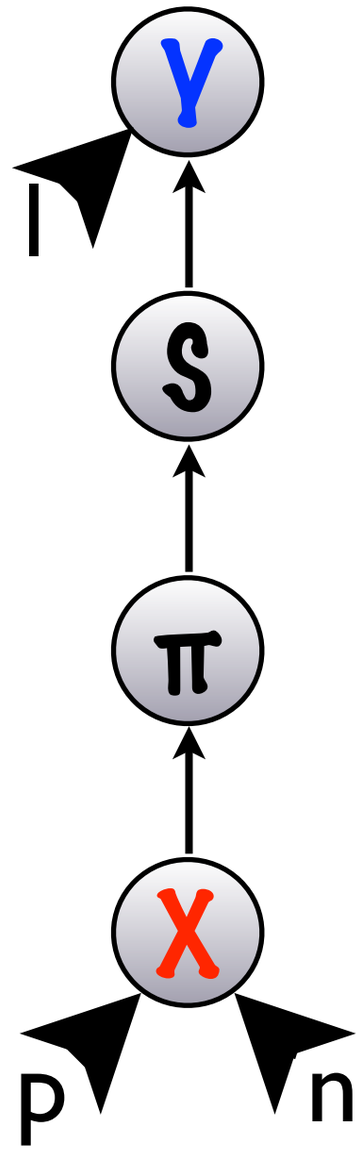
Graph Programming



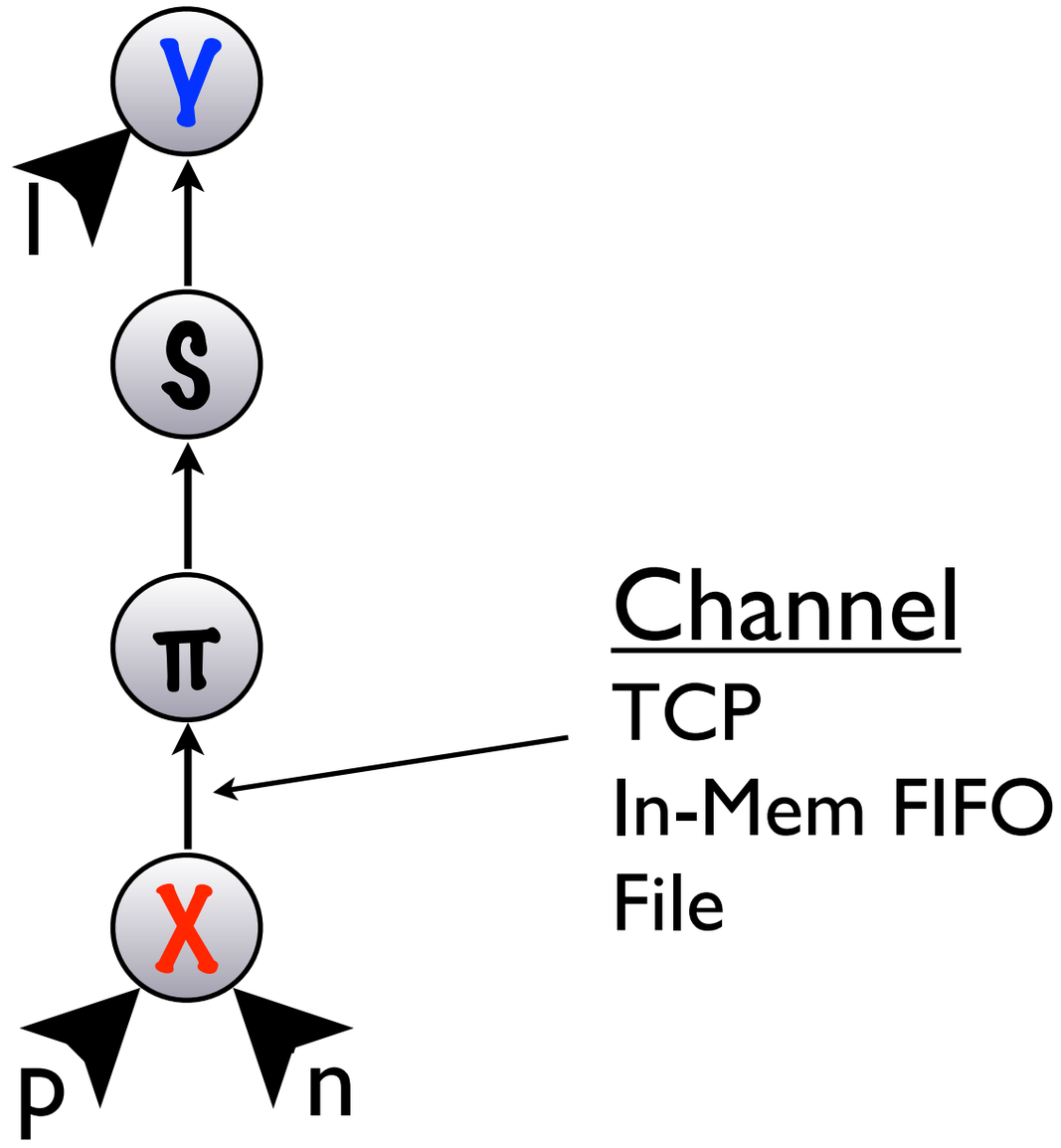

```
Read n.neighborID : p⋈n
Emit sorted p⋈n by n.neighborId
```



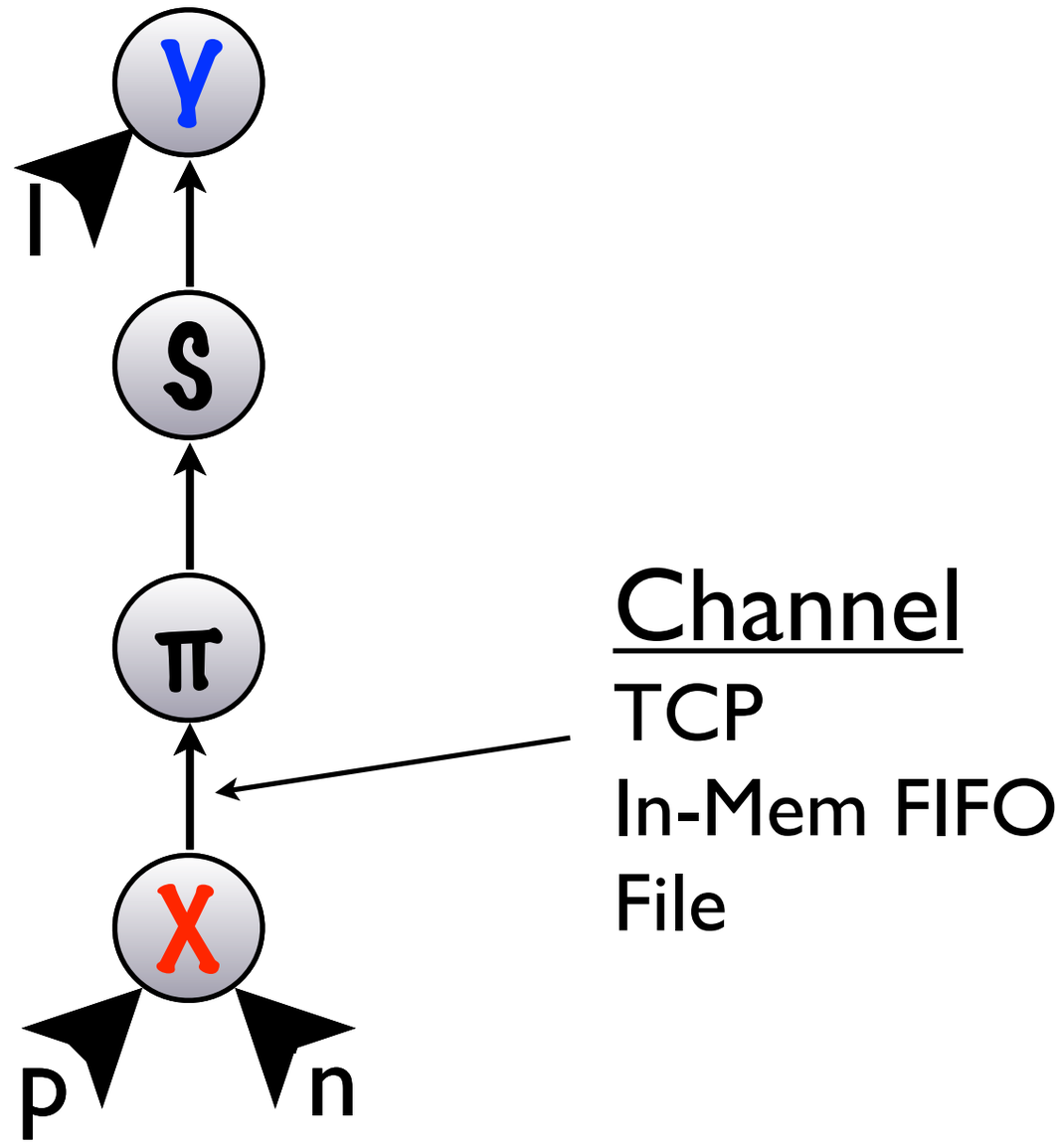
Graph Programming



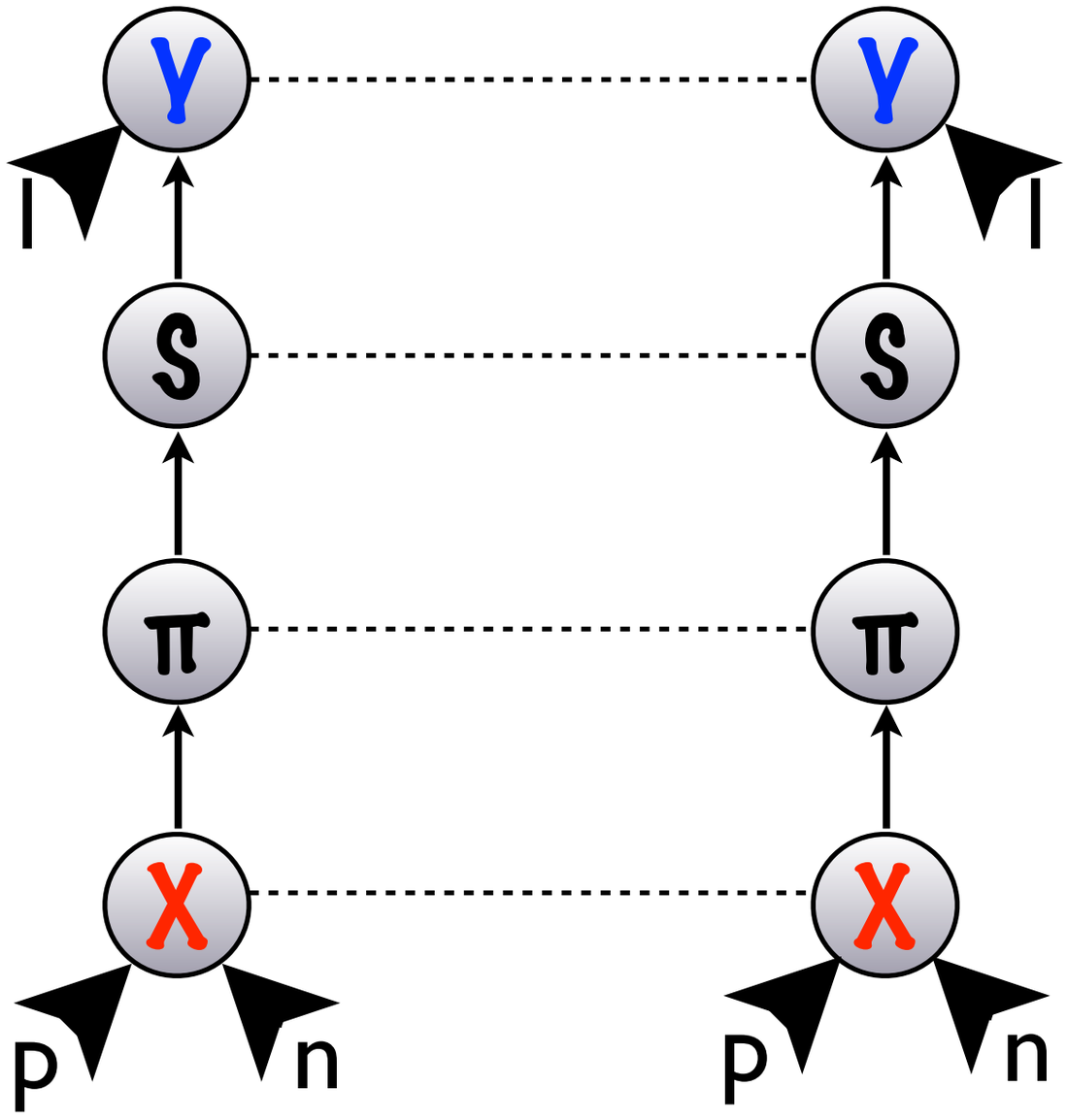
Graph Programming



Graph Programming



Graph Programming



- **Graph Programming**
- Execution Model
- Evaluation

Building a Graph

< V, E, I, O >

Building a Graph

< V, E, I, O >

Building a Graph

< V, E, I, O >

vertices

edges

input vertices

output vertices

Building a Graph

< {A}, {}, {A}, {A} >

Building a Graph

`A^k`

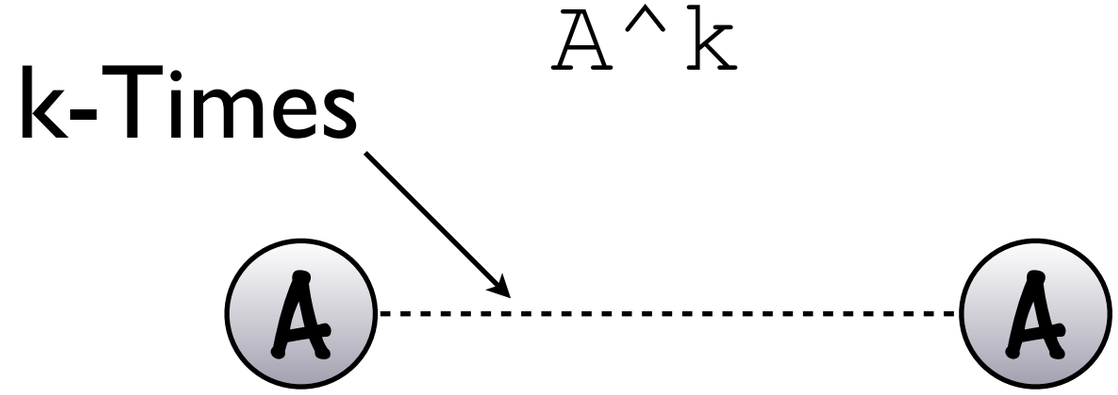
`< {A}, {}, {A}, {A} >`

Building a Graph

`A^k`

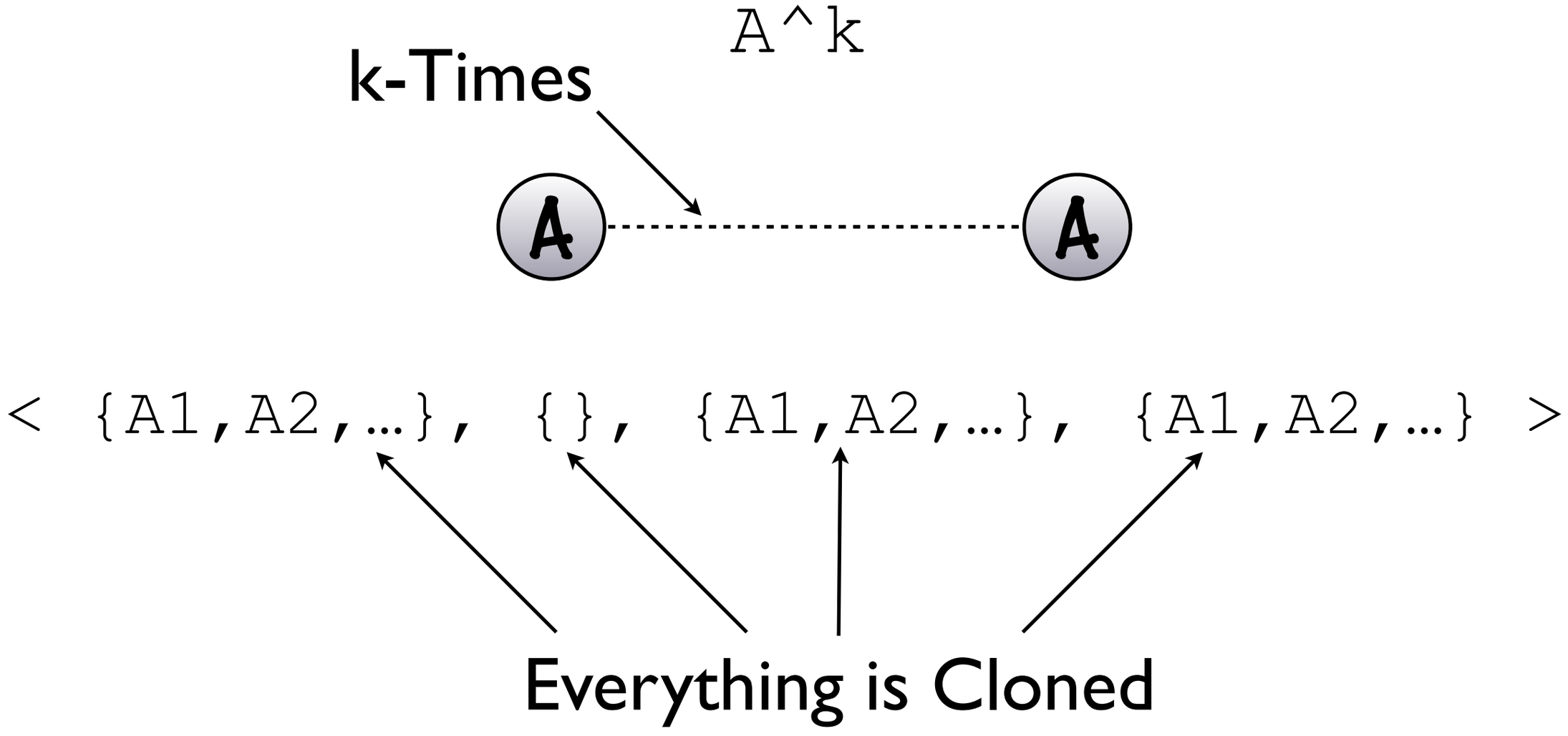
`< {A1,A2,…}, {}, {A1,A2,…}, {A1,A2,…} >`

Building a Graph



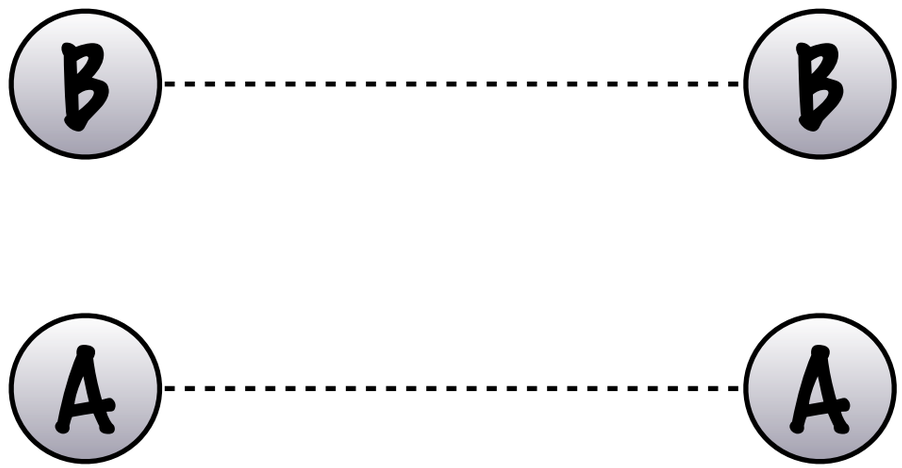

```
< {A1,A2,…}, {}, {A1,A2,…}, {A1,A2,…} >
```

Building a Graph



Building a Graph

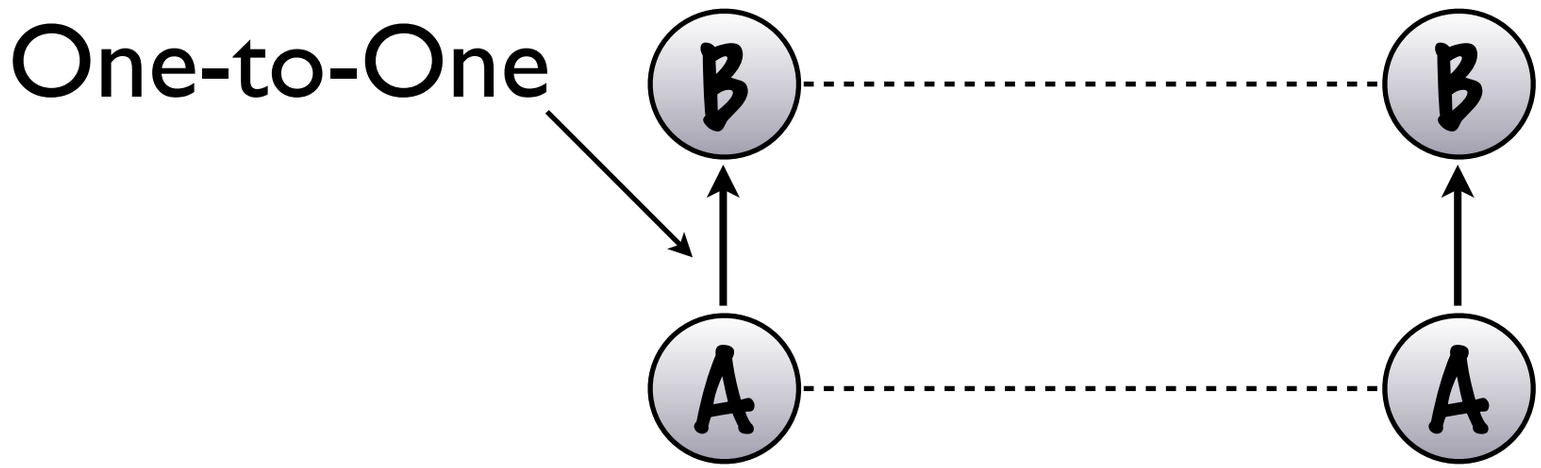
$$A^k >= B^k$$



$$< V_A \oplus V_B,\ E_A \cup E_B \cup E_{new},\ I_A,\ O_B >$$

Building a Graph

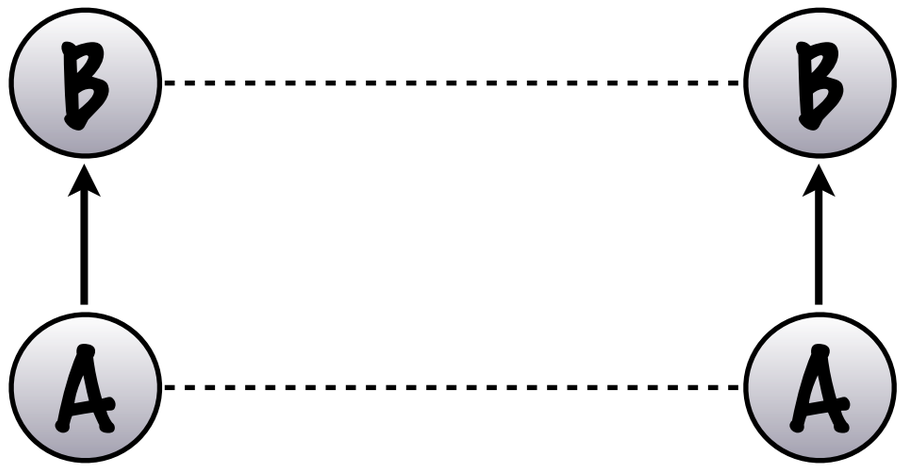
$$A^k >= B^k$$



$$< V_A \oplus V_B,\ E_A \cup E_B \cup E_{new},\ I_A,\ O_B >$$

Building a Graph

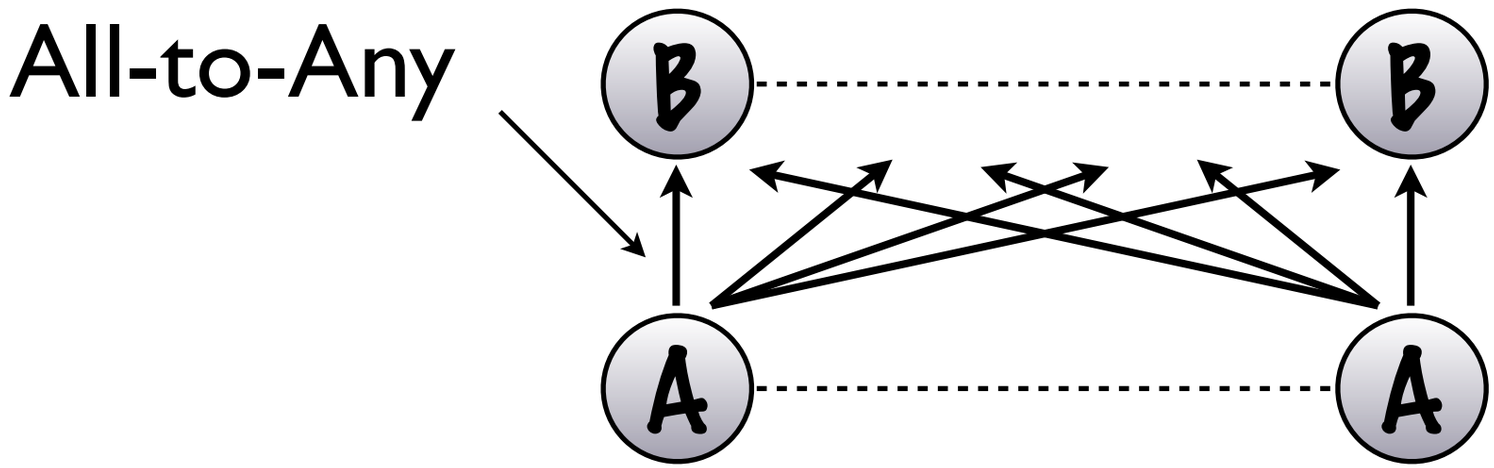
$$A^k >= B^k$$



$$< V_A \oplus V_B,\ E_A \cup E_B \cup E_{new},\ I_A,\ O_B >$$

Building a Graph

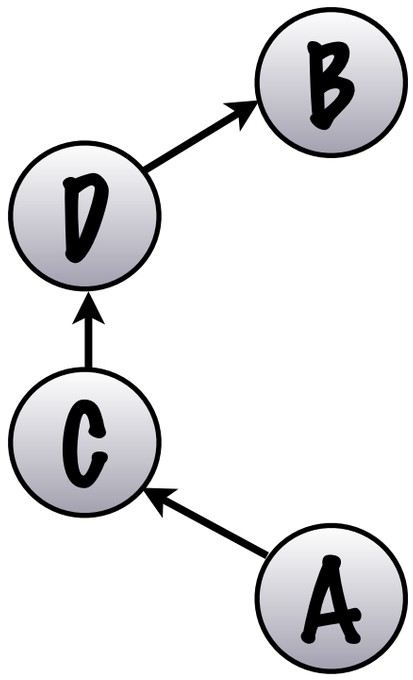
`A^k >> B^k`

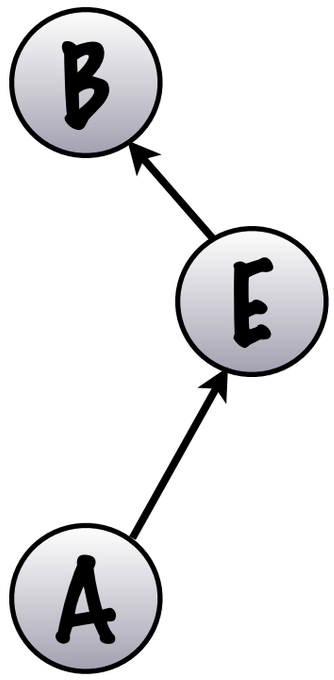


$< V_A \oplus V_B,\ E_A \cup E_B \cup E_{new},\ I_A,\ O_B >$

Building a Graph

(A>=C>=D>=B) || (A>=E=>B)

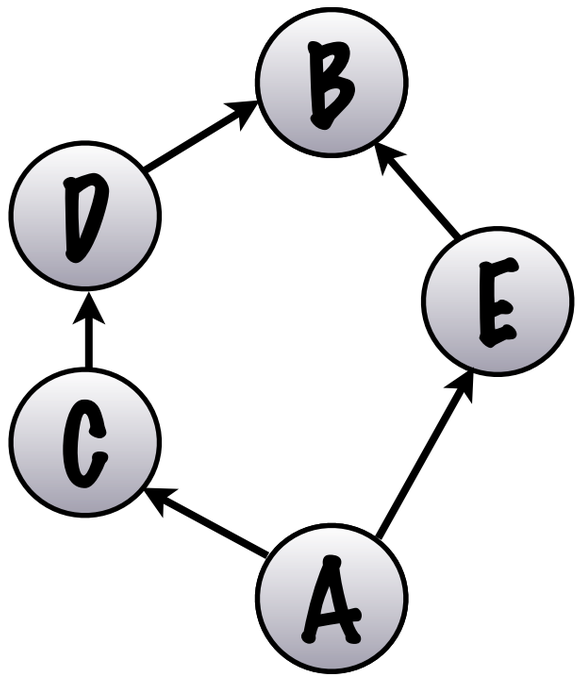




$< V_A \oplus V_B,\ E_A \cup E_B,\ I_A \cup I_B,\ O_A \cup O_B >$

Building a Graph

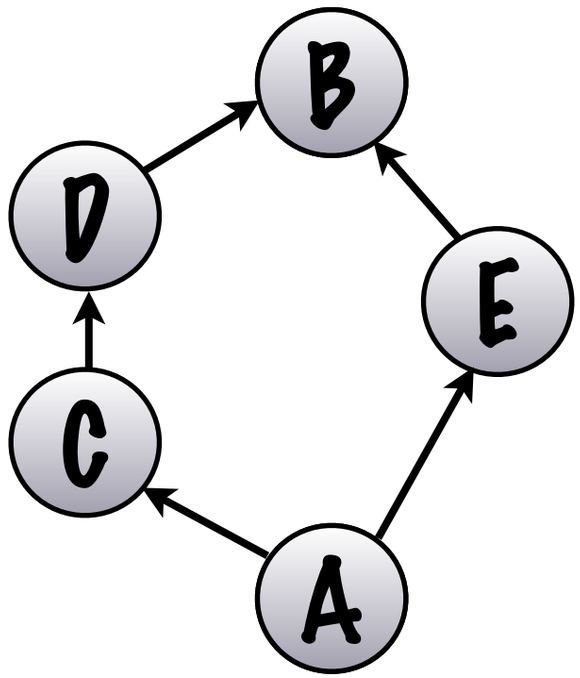
(A>=C>=D>=B) || (A>=E=>B)



$< V_A \oplus V_B,\ E_A \cup E_B,\ I_A \cup I_B,\ O_A \cup O_B >$

Building a Graph

$$(A>=C>=D>=B) \;||\; (A>=E=>B)$$



$$< V_A \oplus^{*} V_B,\; E_A \cup^{*} E_B,\; I_A \cup^{*} I_B,\; O_A \cup^{*} O_B >$$

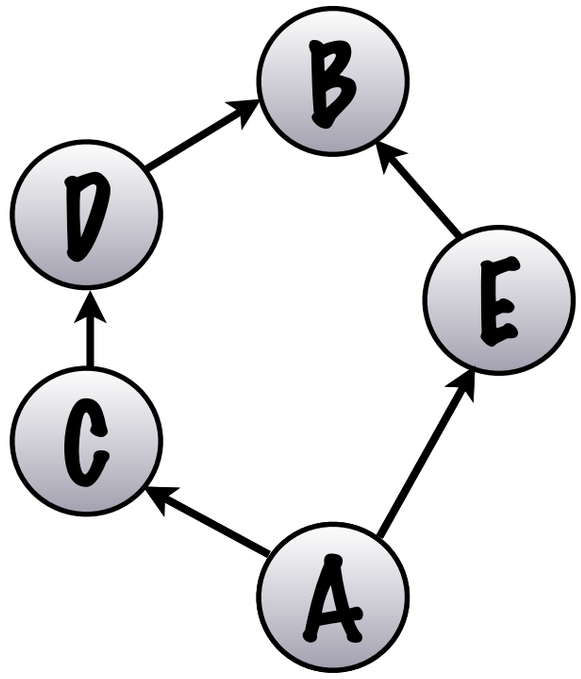
* = except merging duplicates

Summary

`A^k`	Parallelize A, with k replicas
`A >= B`	Connect A's outputs to B's inputs (one-to-one)
`A >> B`	Connect A's outputs to B's inputs (all-to-any)
`A || B`	Merge graphs A and B (deduplicating nodes in both A and B)

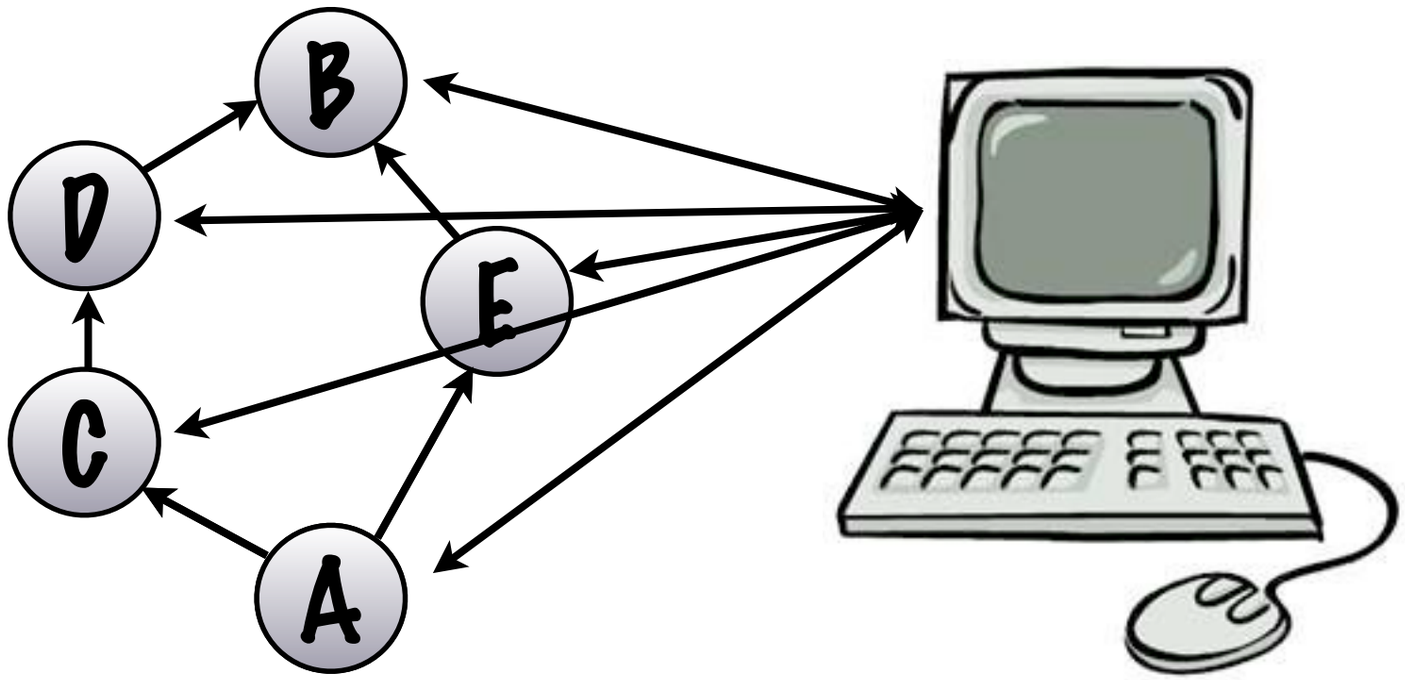
- Graph Programming
- **Execution Model**
- Evaluation

Job Execution



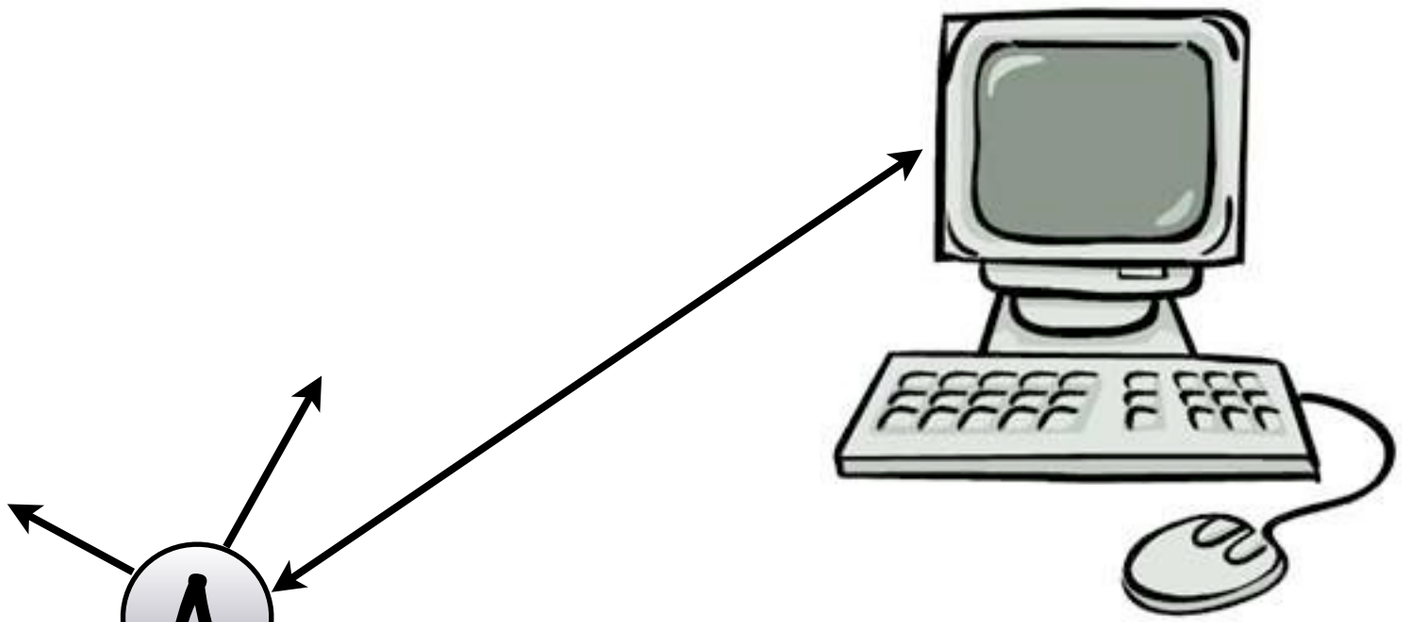
Job Execution

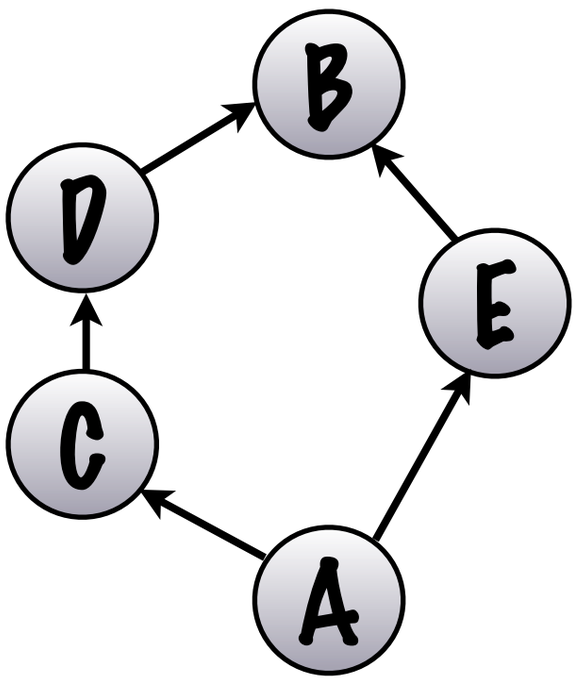
Job Manager



The Job Manager coordinates graph execution

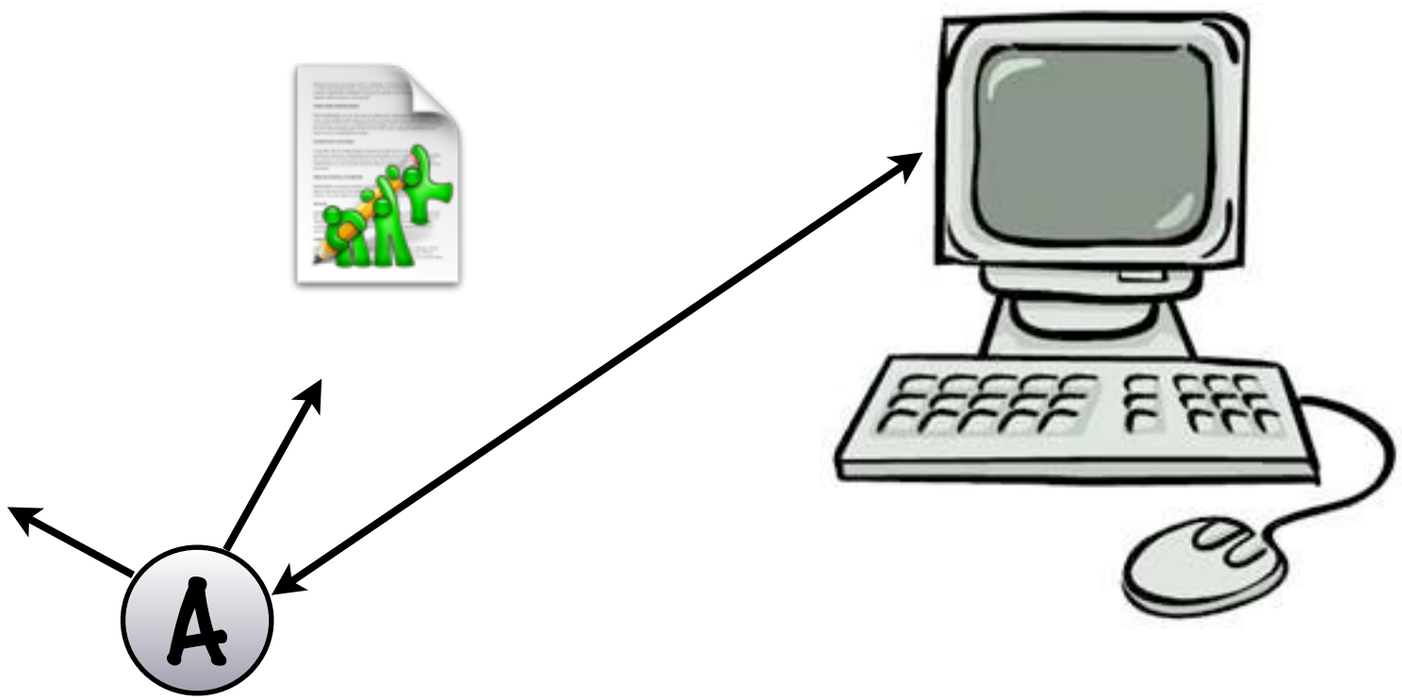
Job Execution

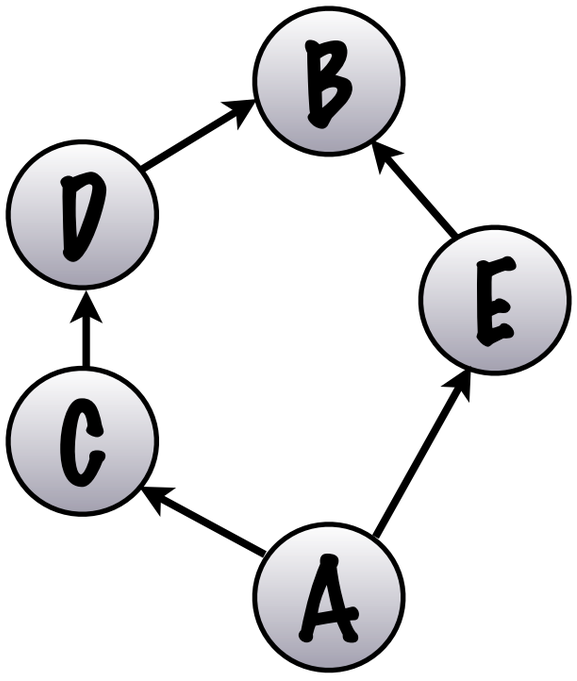




Edges can be implemented as...

Job Execution

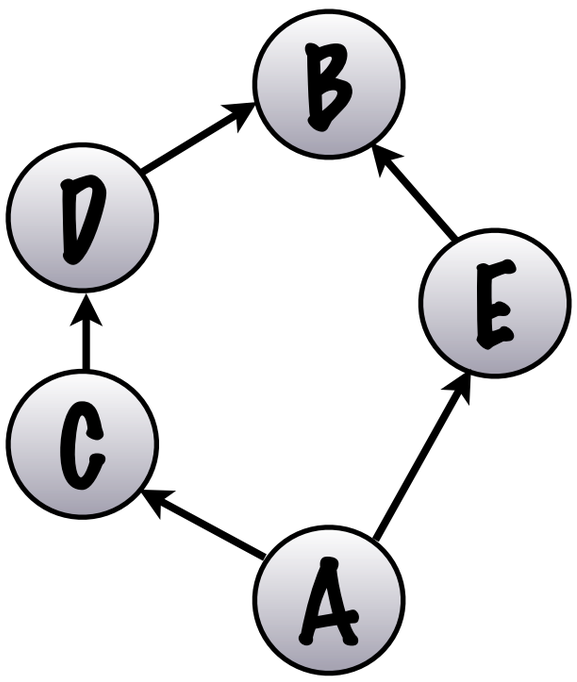




Edges can be implemented as...

Files,

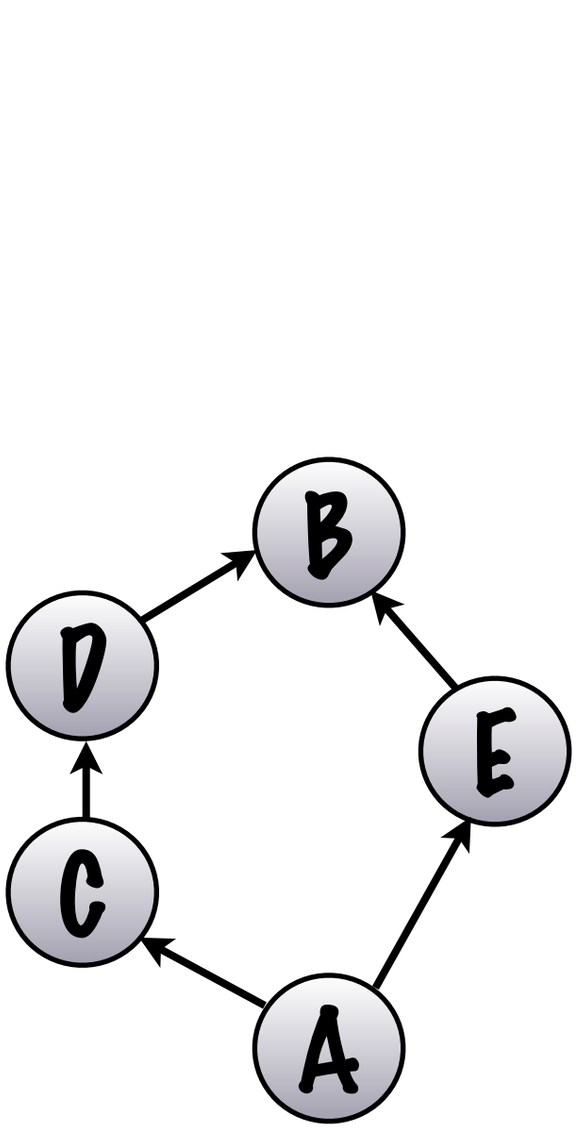
Job Execution

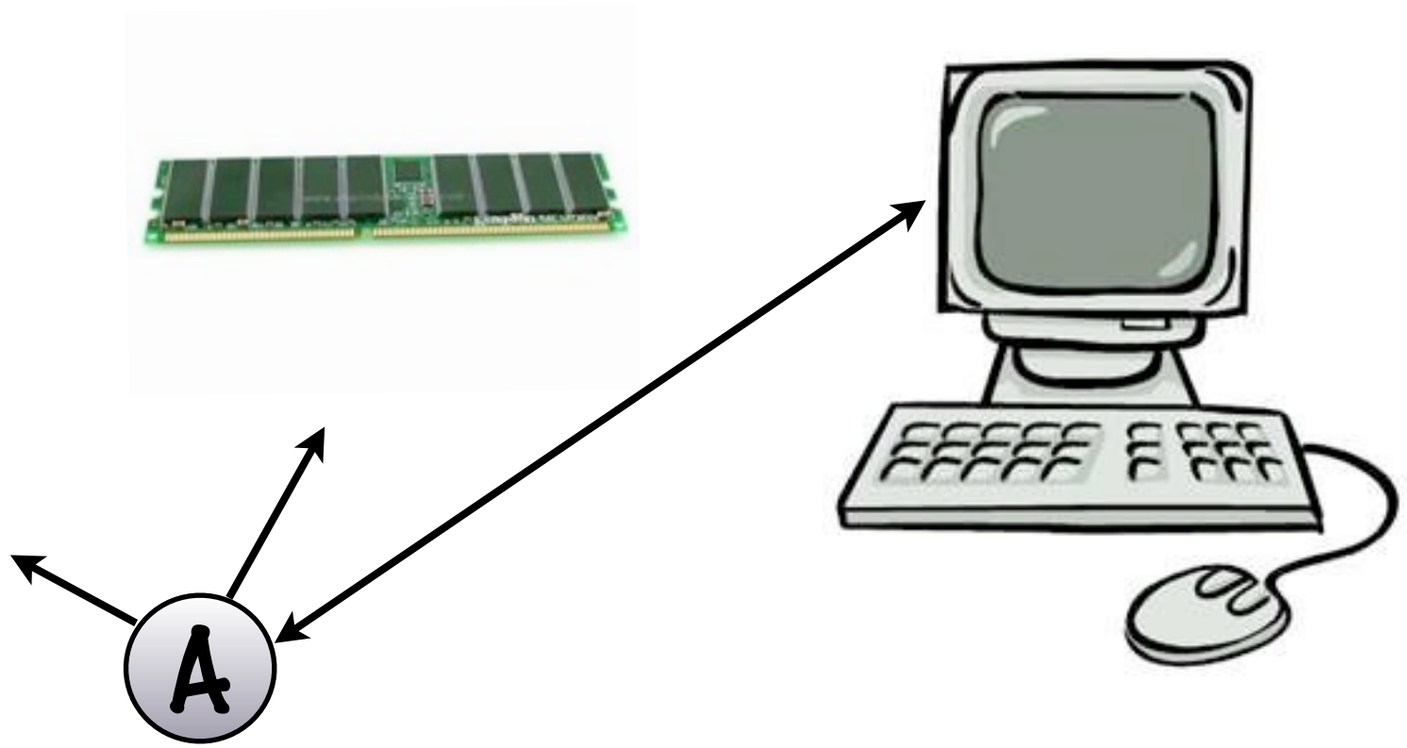


Edges can be implemented as...

Files, TCP Streams,

Job Execution

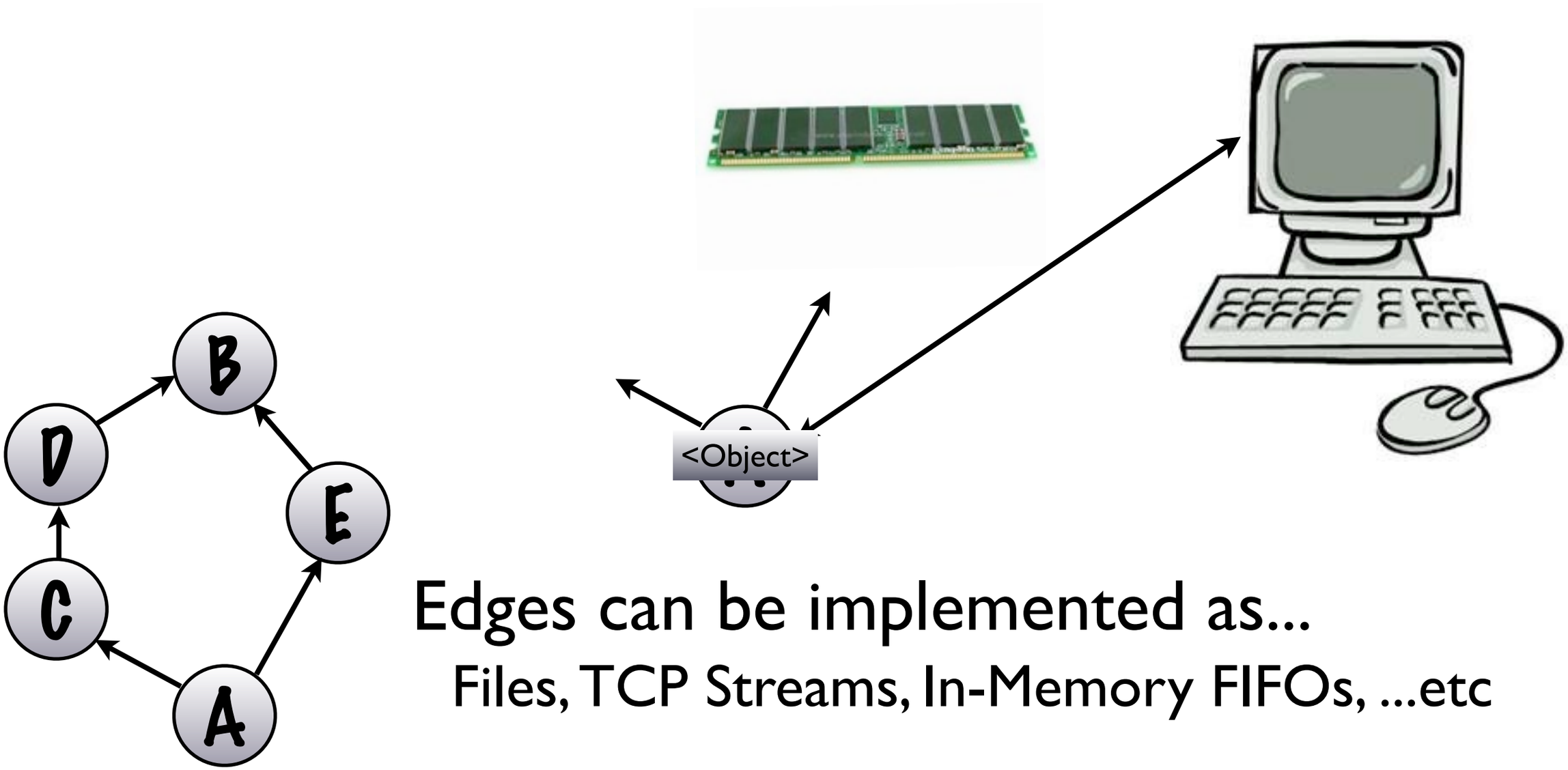




Edges can be implemented as...

Files, TCP Streams, In-Memory FIFOs, ...etc

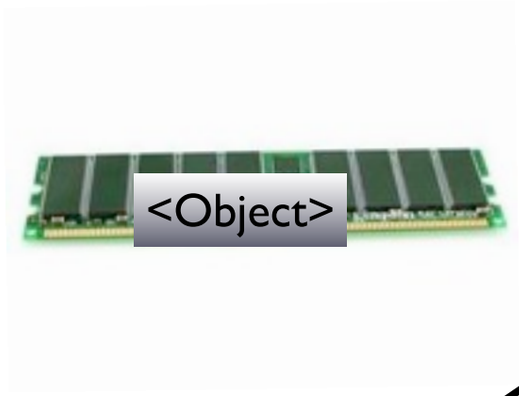
Job Execution

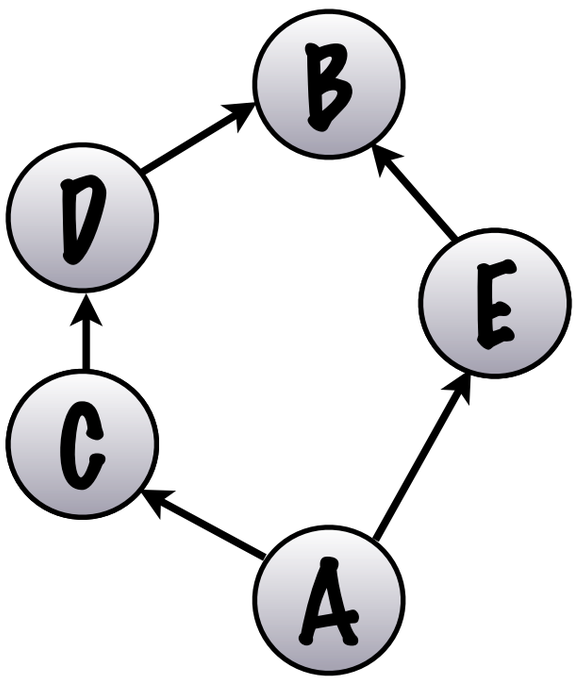


Edges can be implemented as...

Files, TCP Streams, In-Memory FIFOs, ...etc

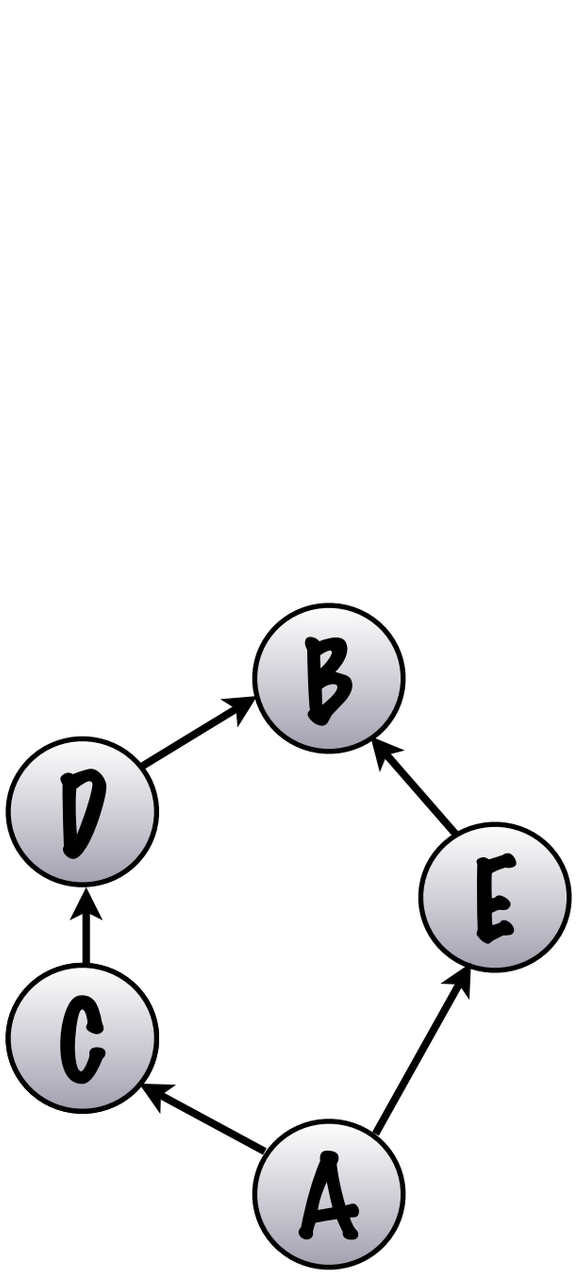
Job Execution

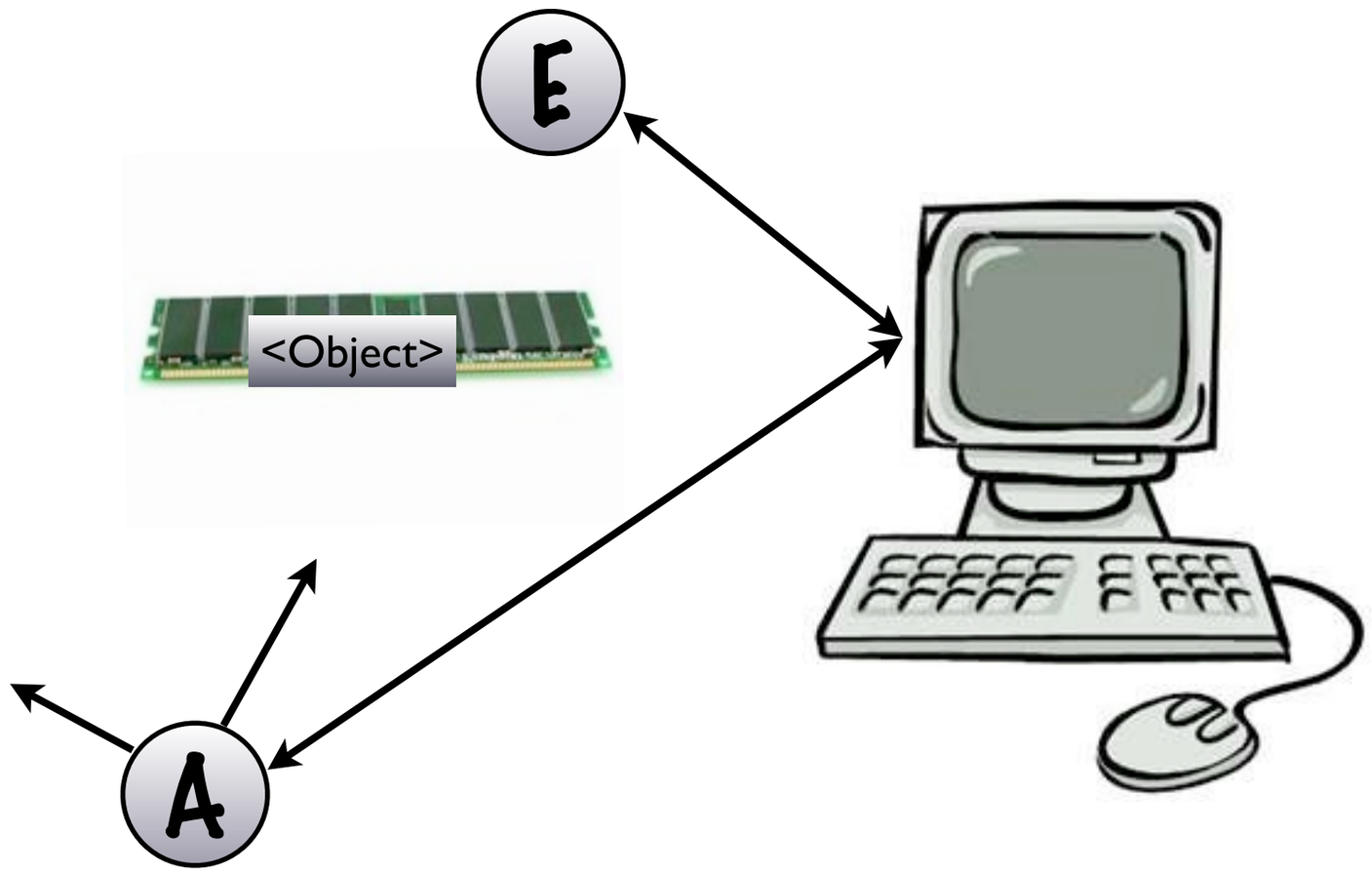




The programmer doesn't need to worry about the channels

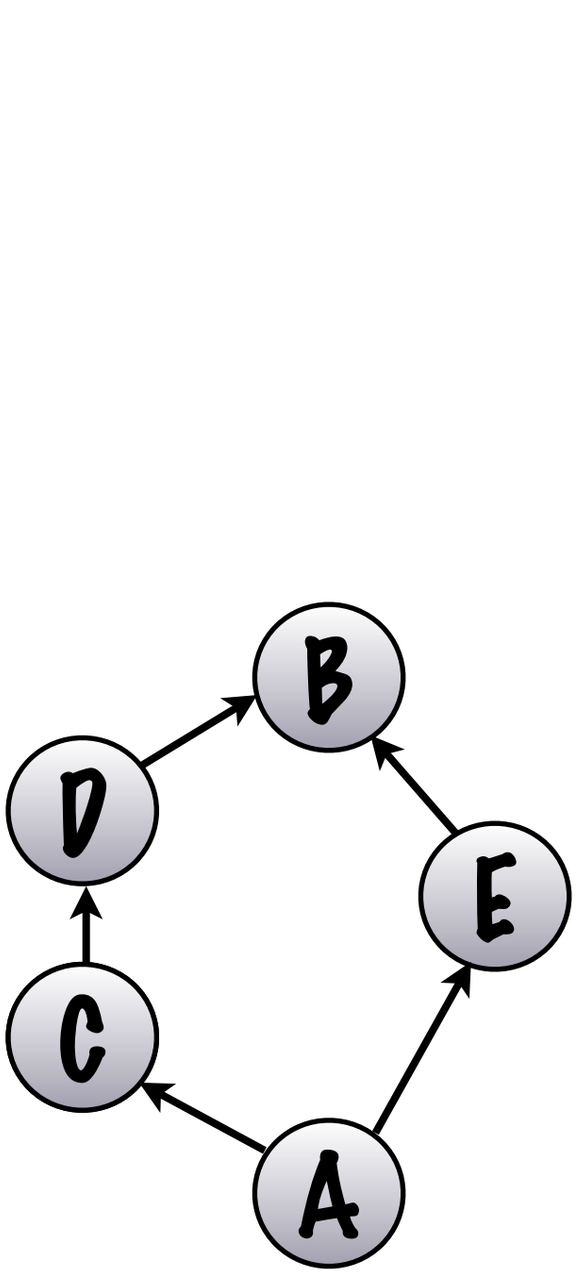
Job Execution

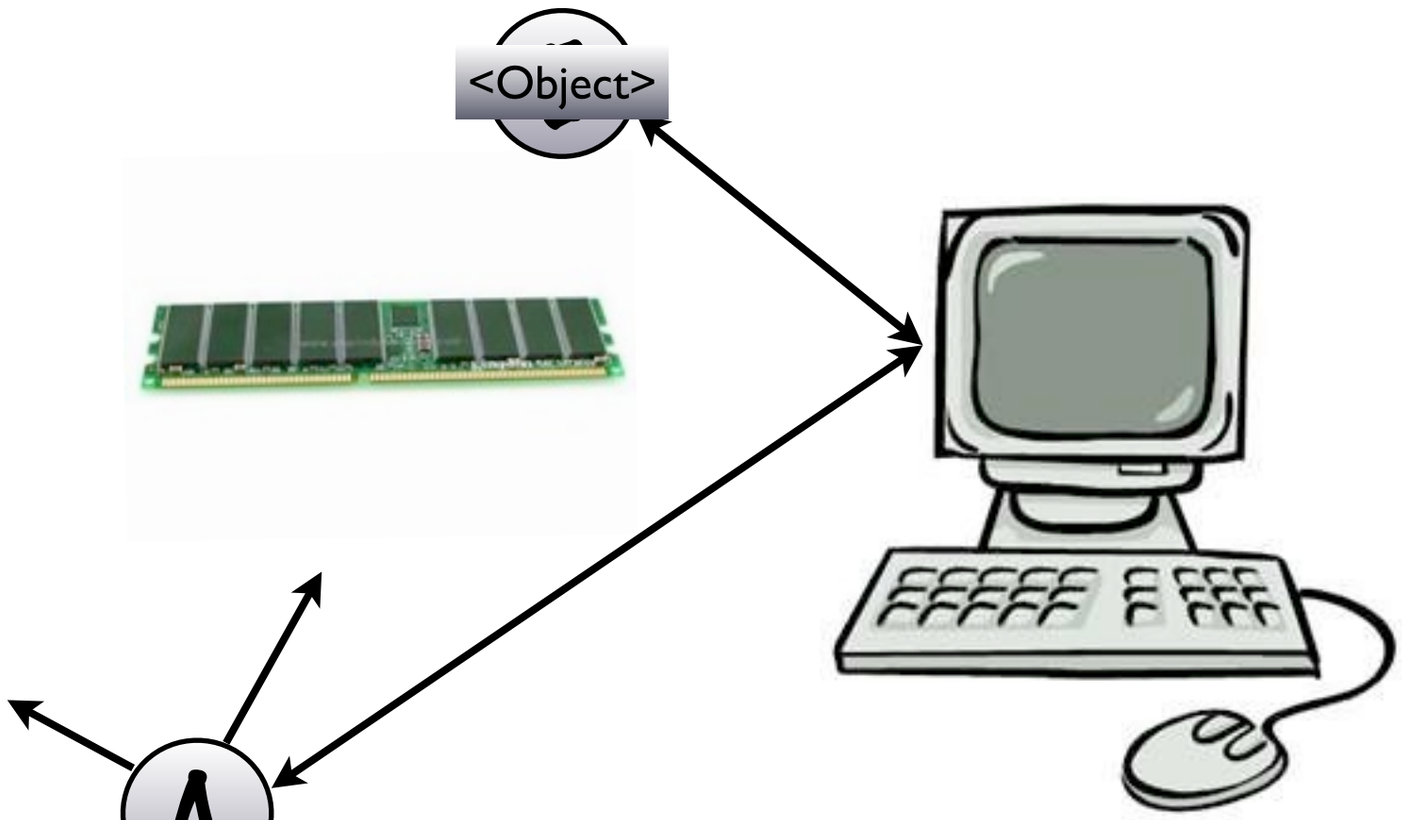




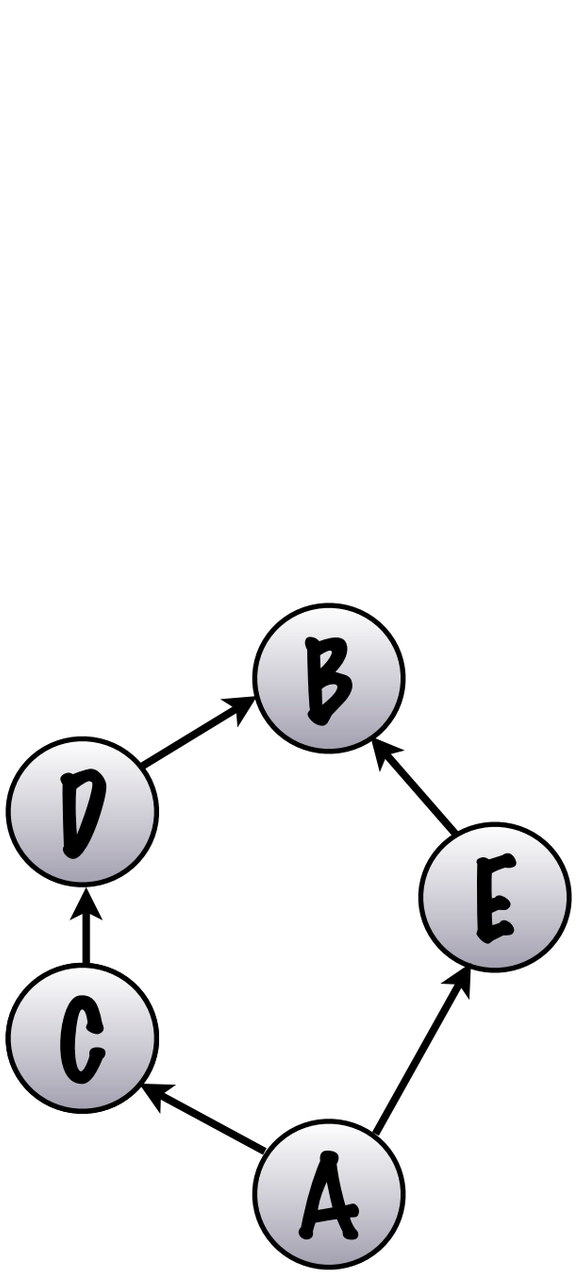
For In-Memory FIFOs, downstream nodes are spawned immediately.

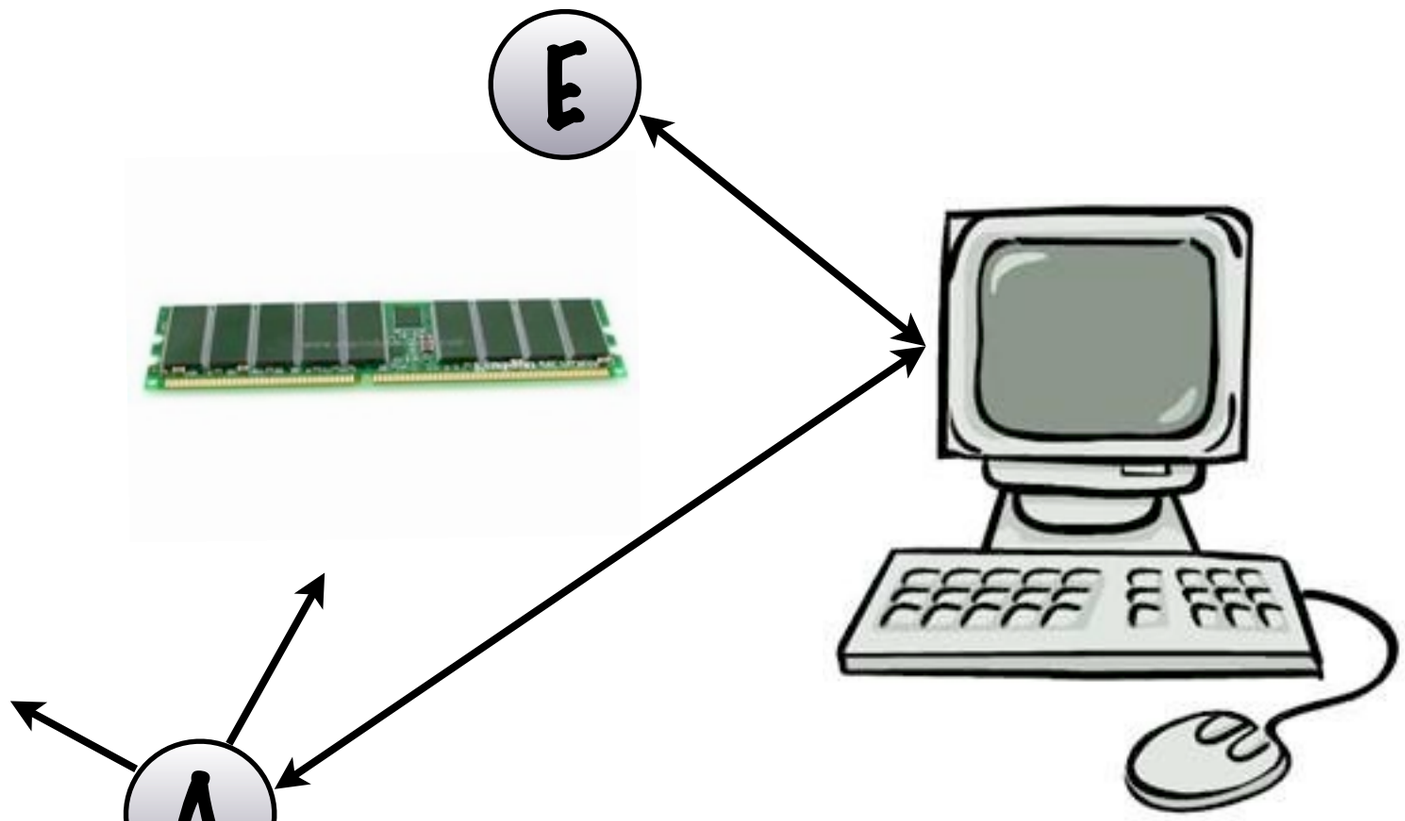
Job Execution



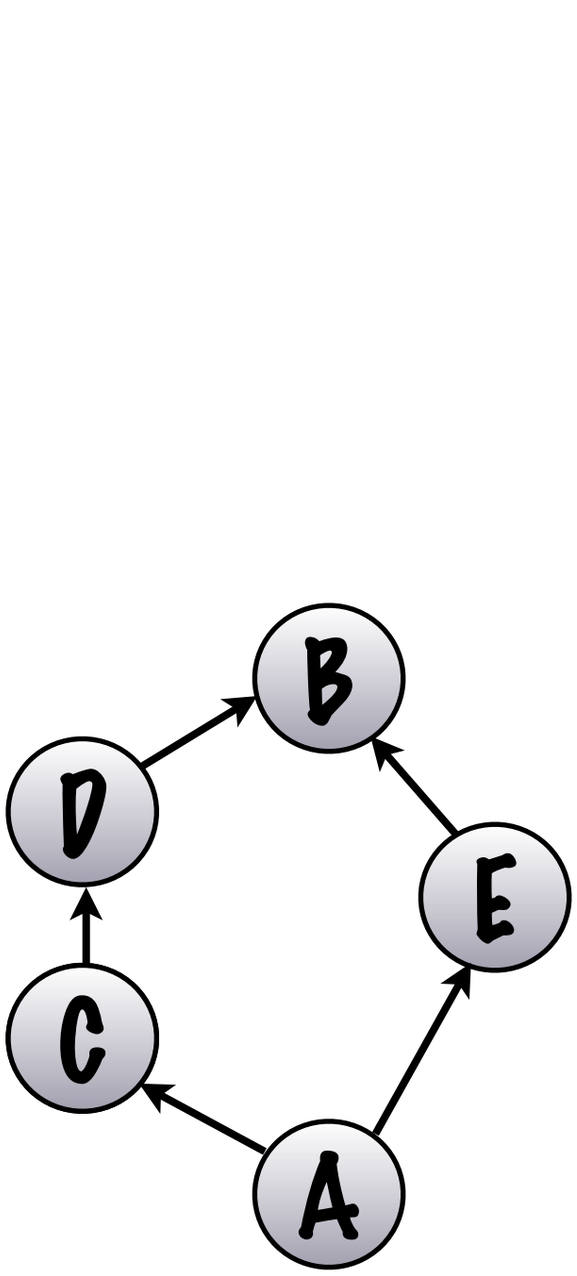


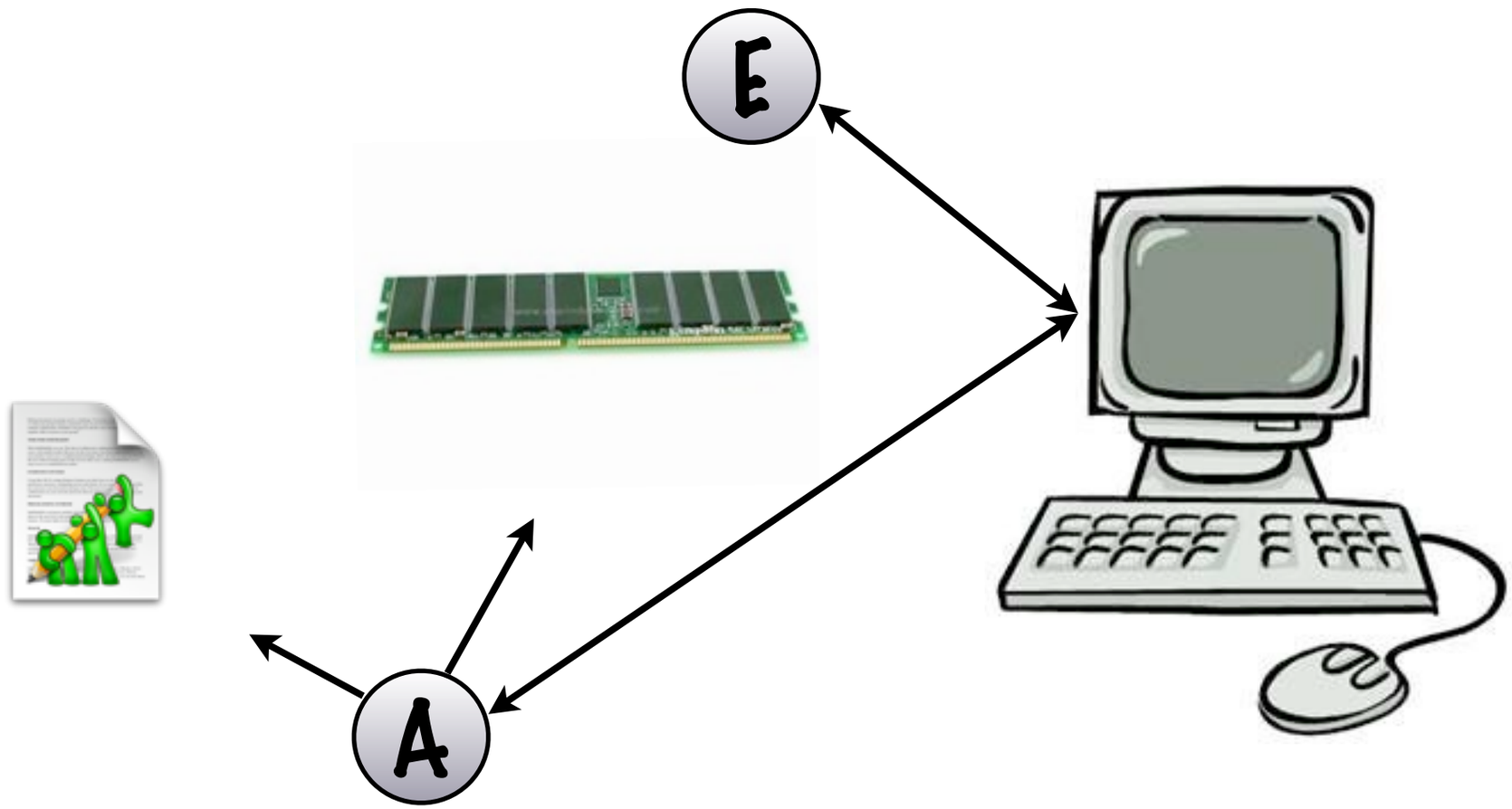
Job Execution





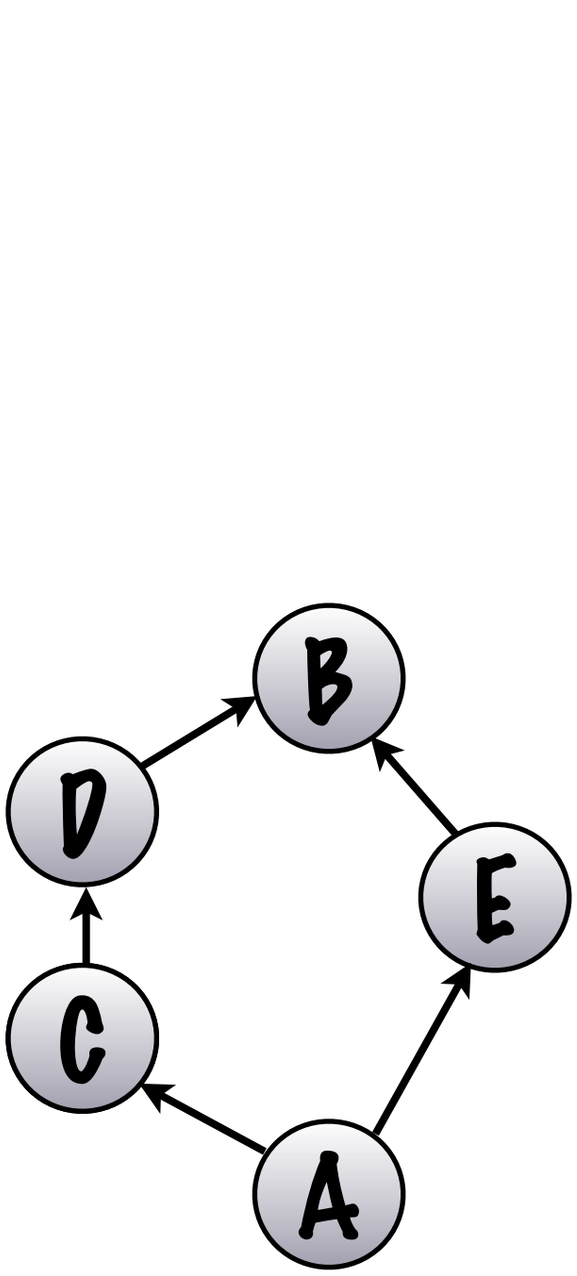
Job Execution

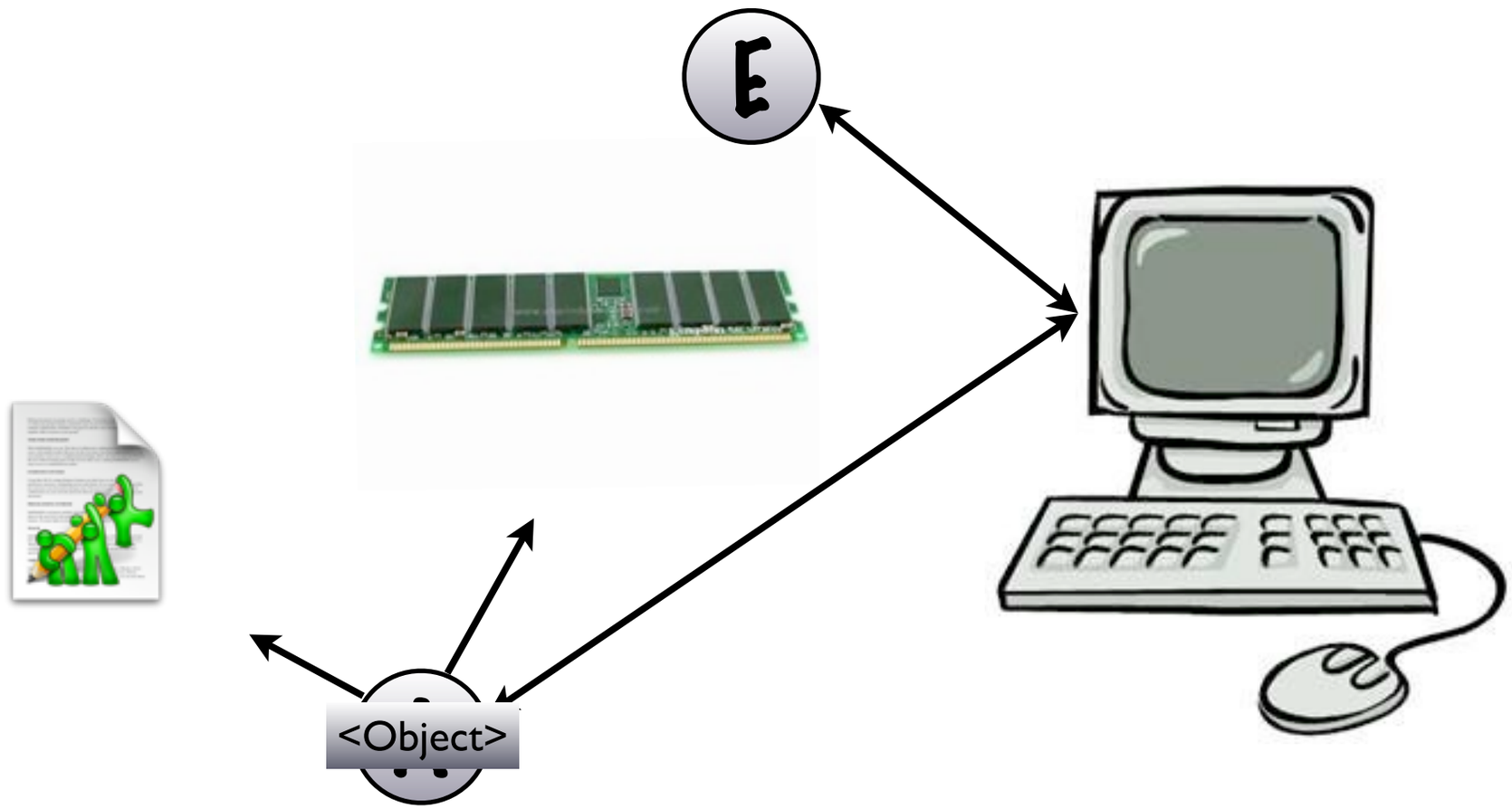




Files are completed before spawning downstream nodes.

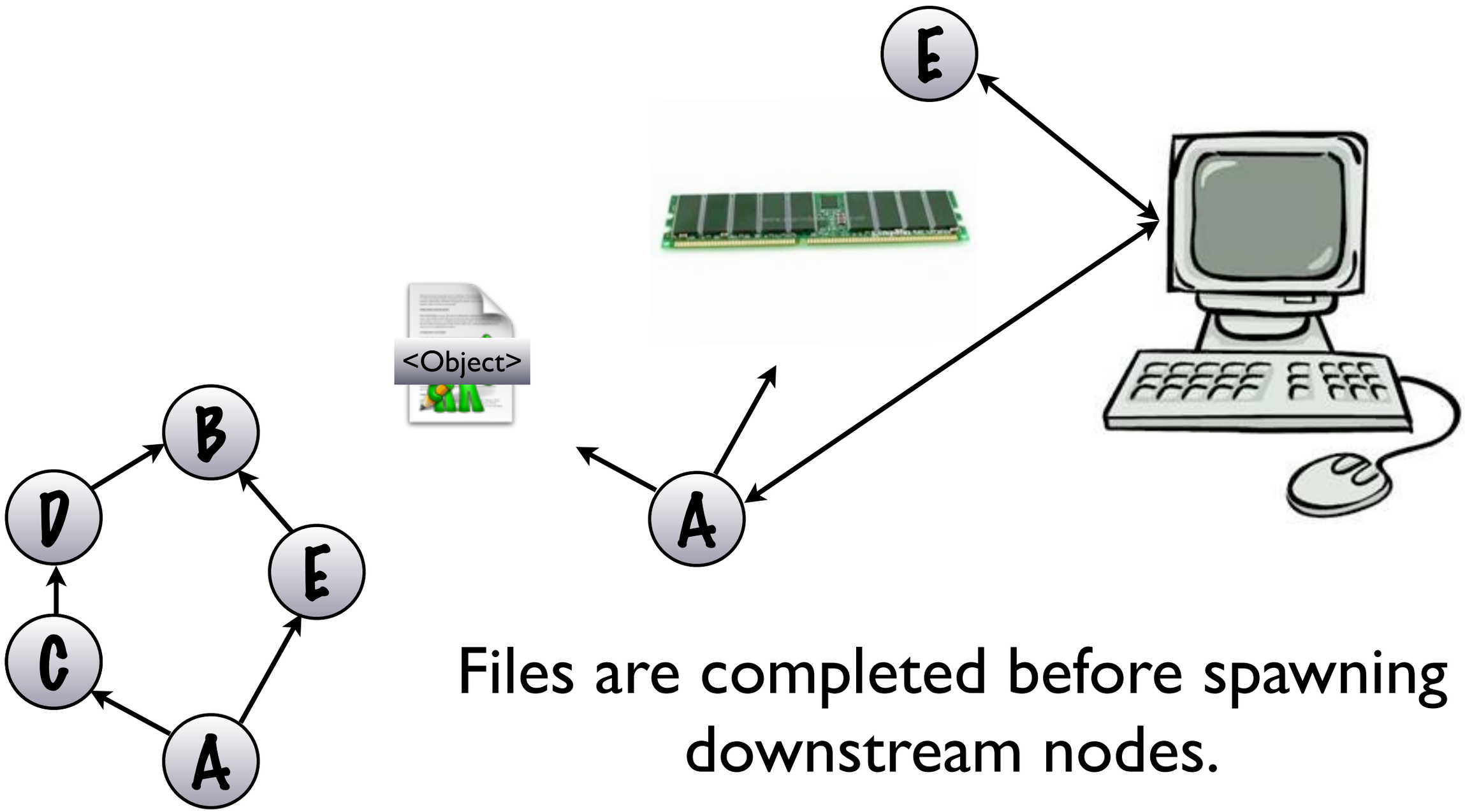
Job Execution





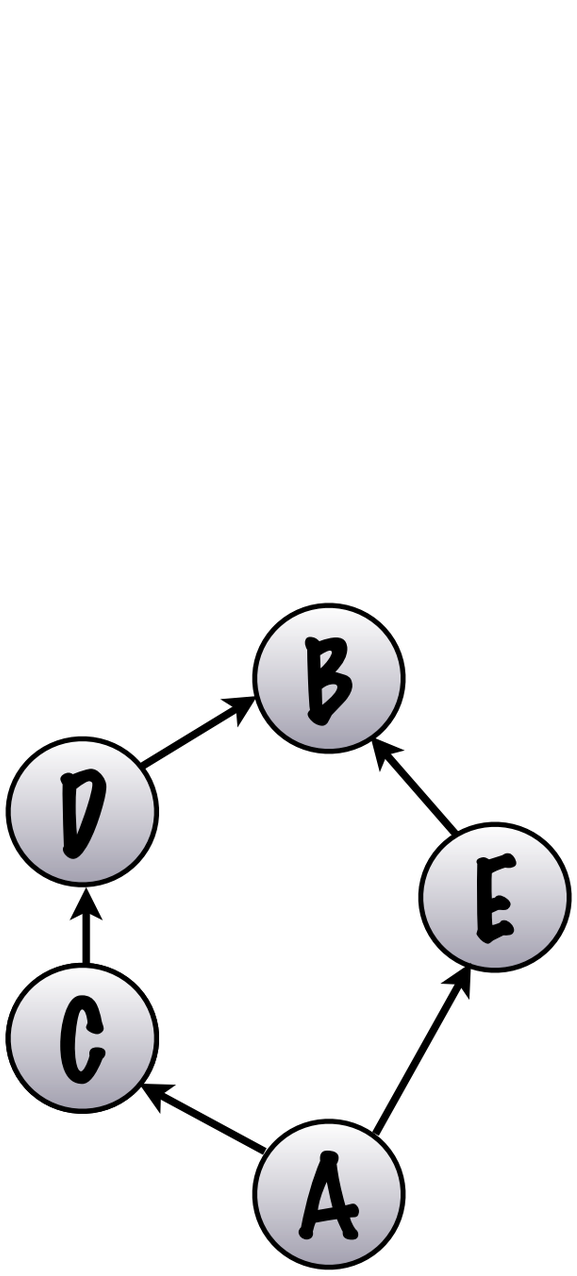
Files are completed before spawning downstream nodes.

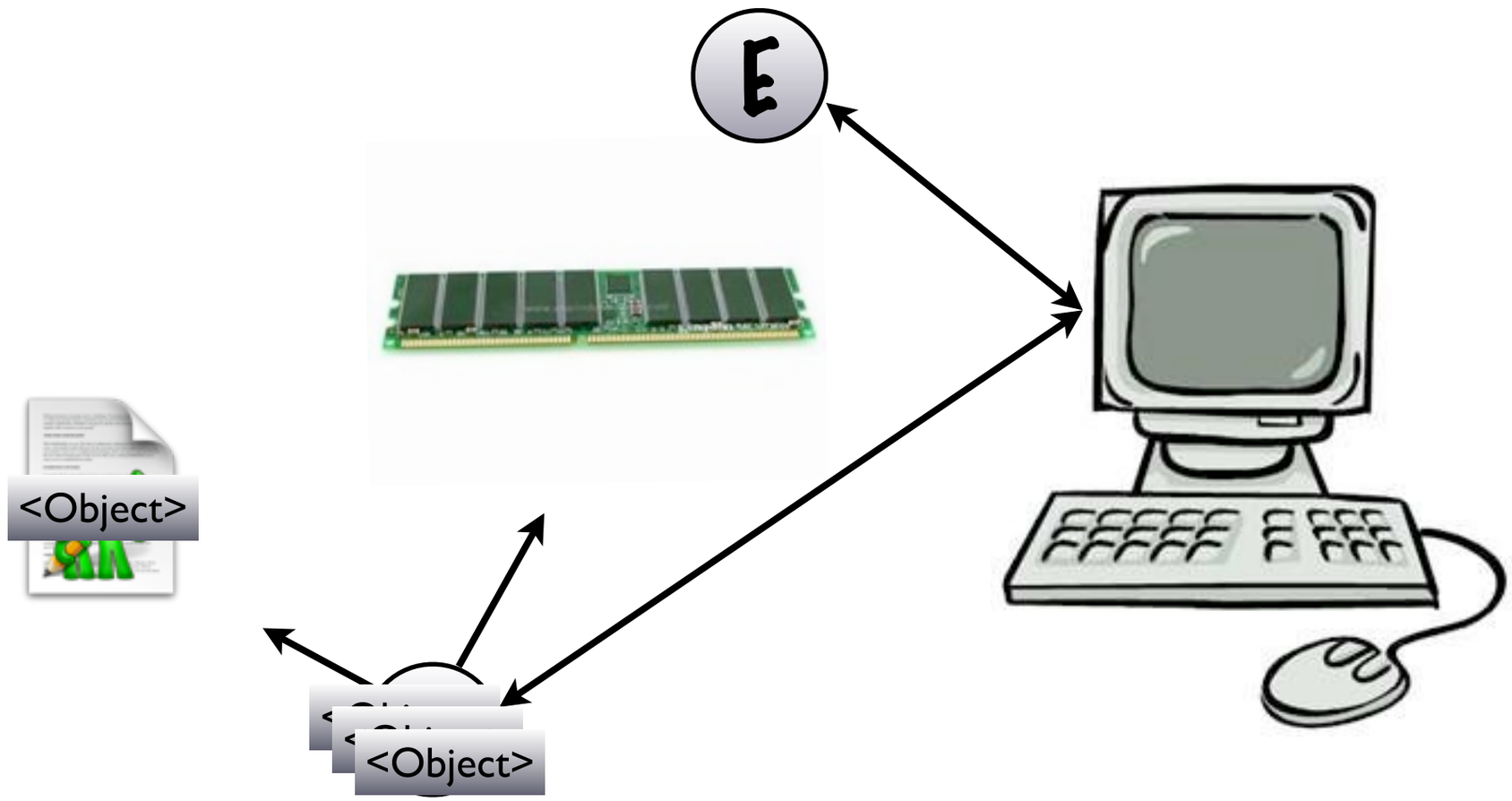
Job Execution



Files are completed before spawning downstream nodes.

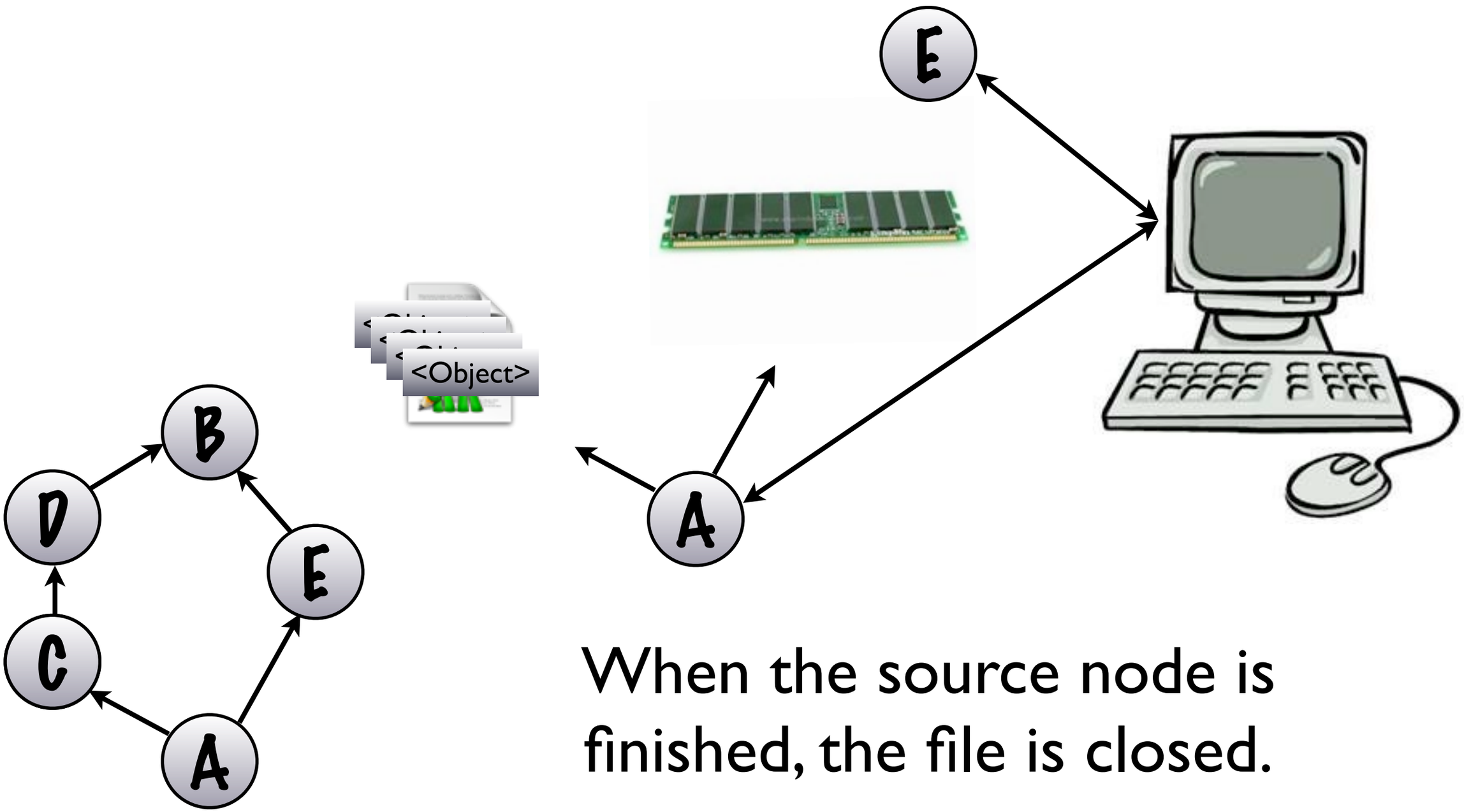
Job Execution





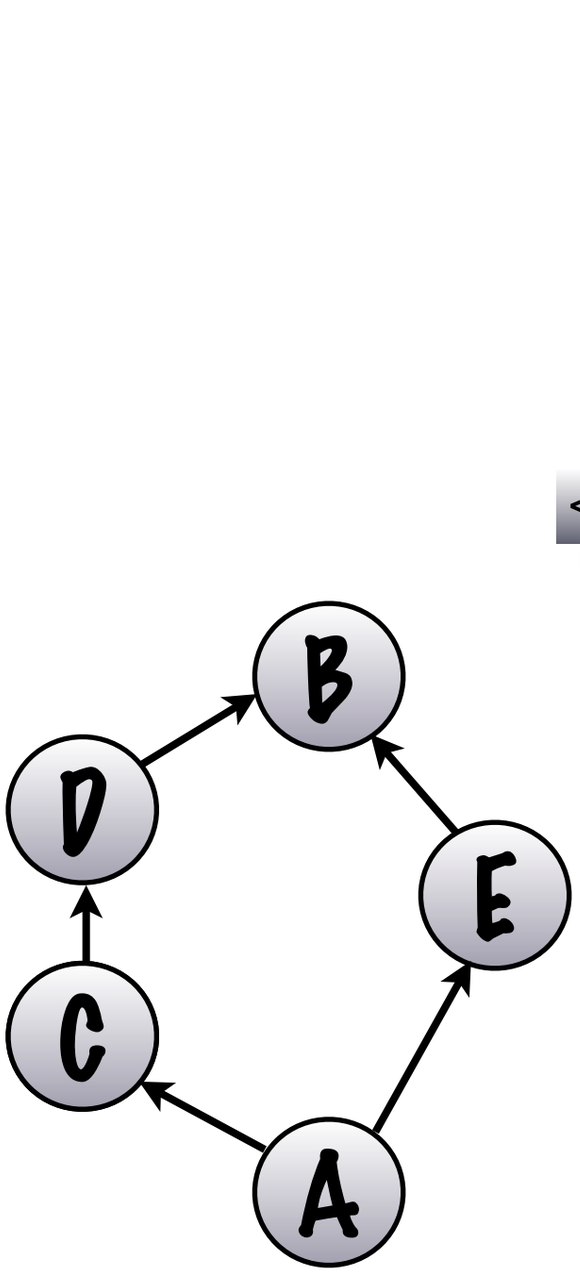
Files are completed before spawning downstream nodes.

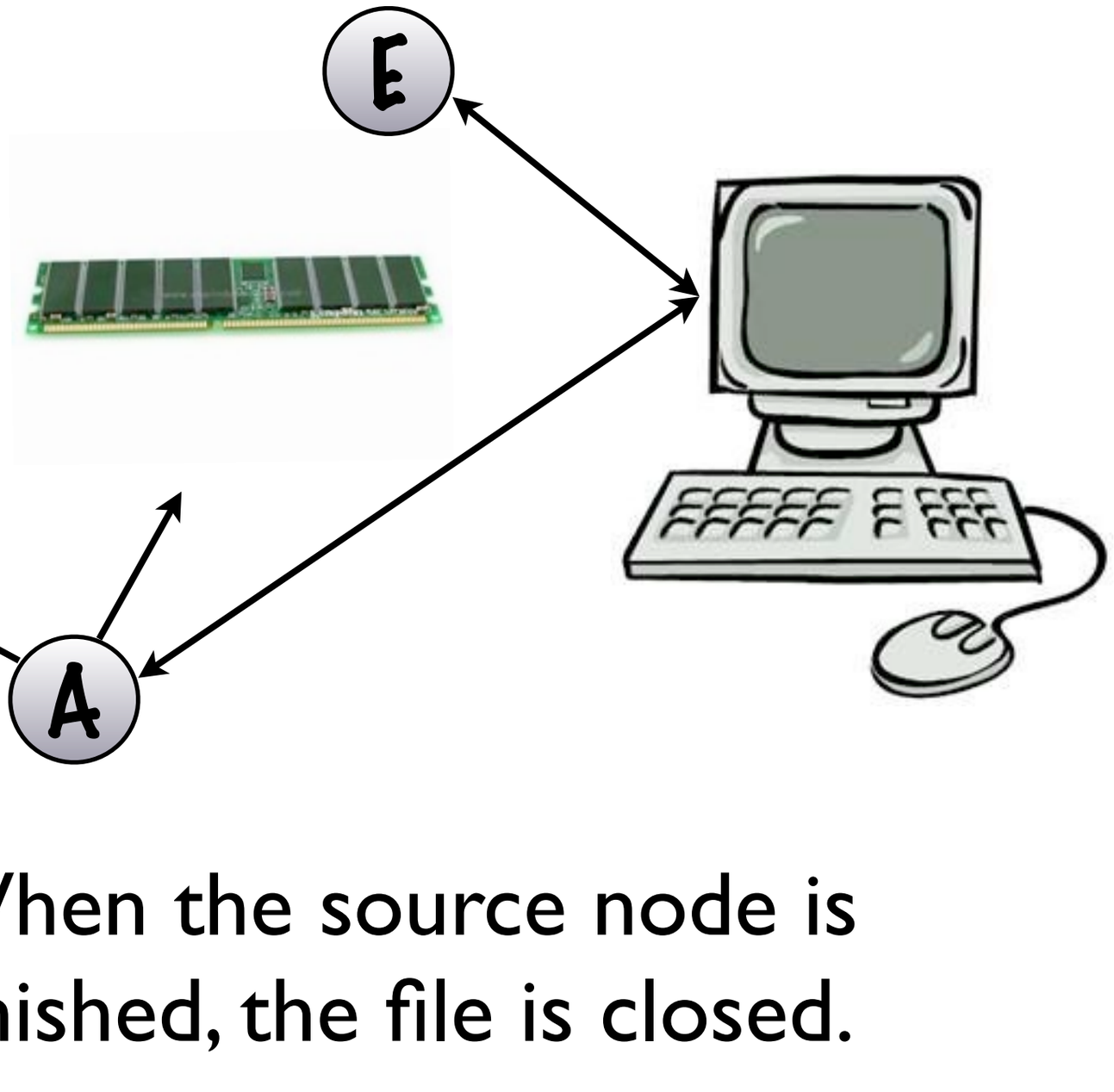
Job Execution



When the source node is finished, the file is closed.

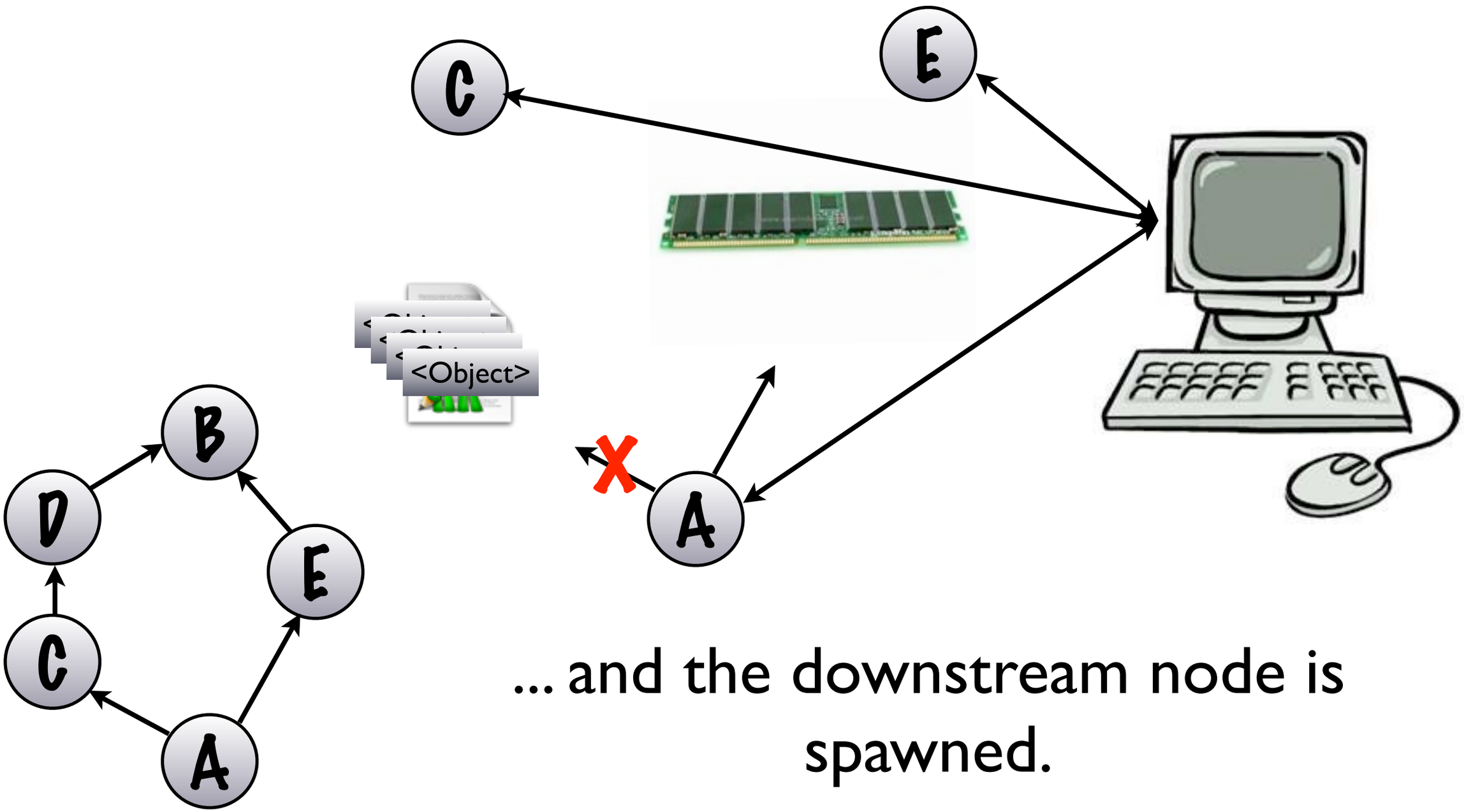
Job Execution





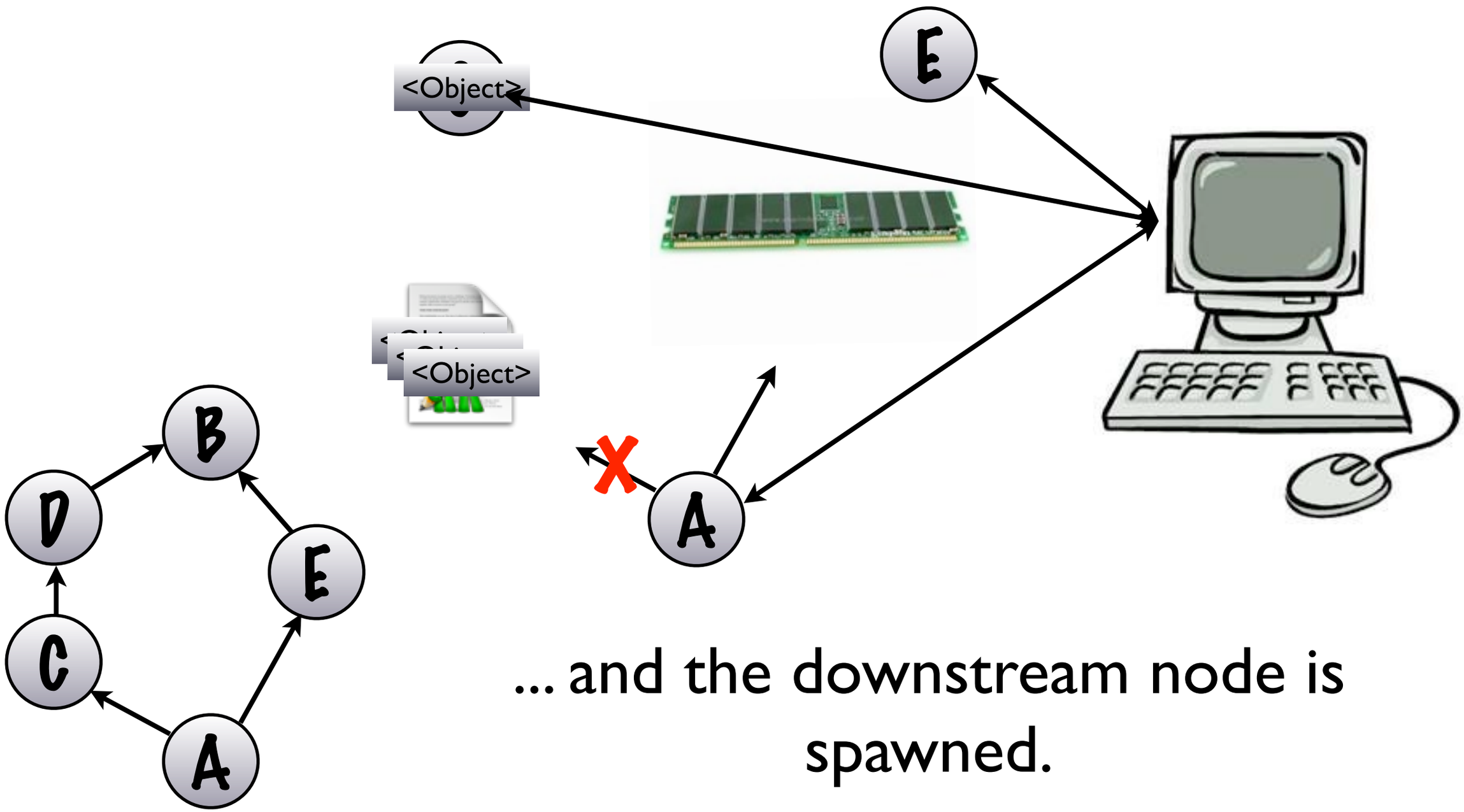
When the source node is finished, the file is closed.

Job Execution



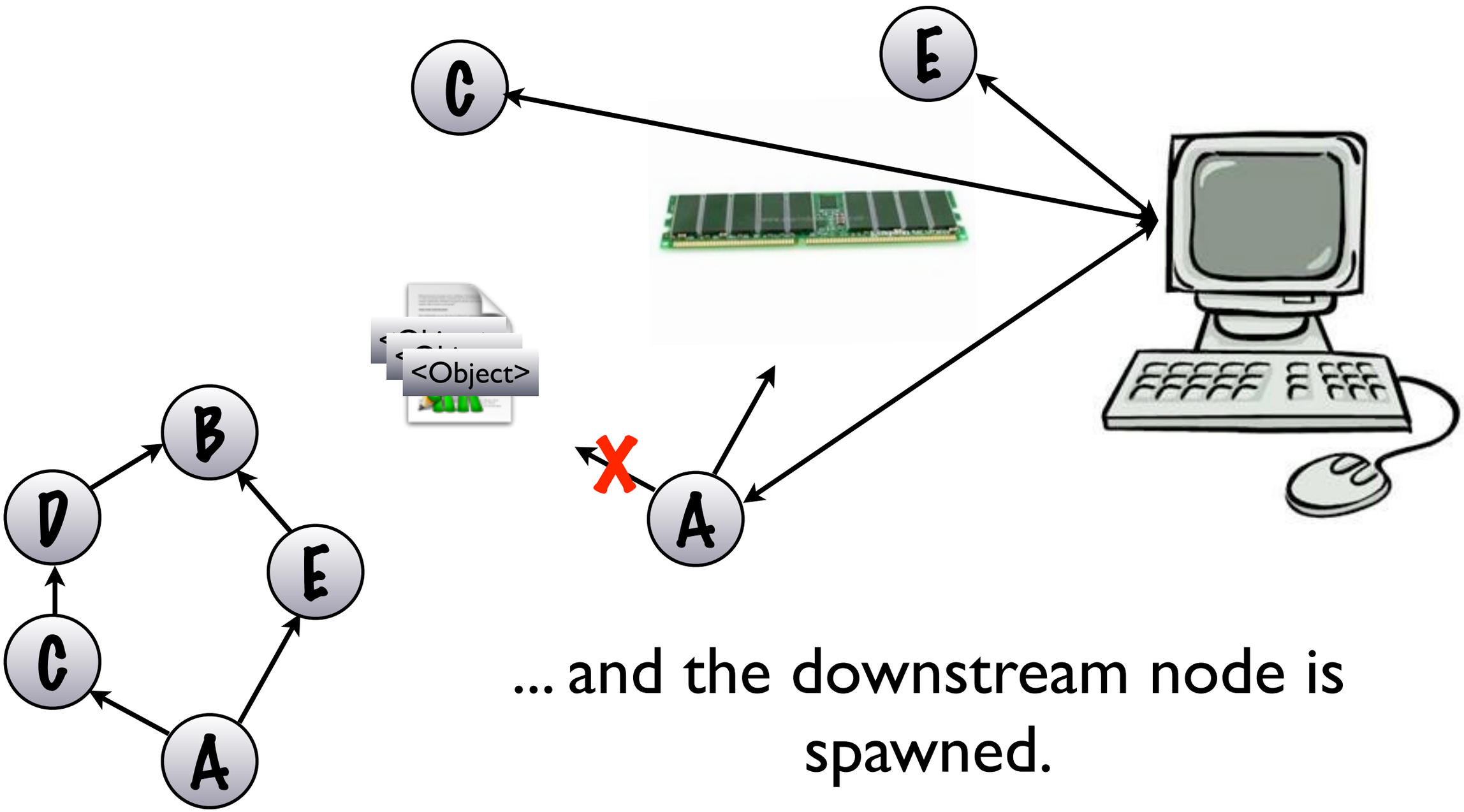
... and the downstream node is spawned.

Job Execution



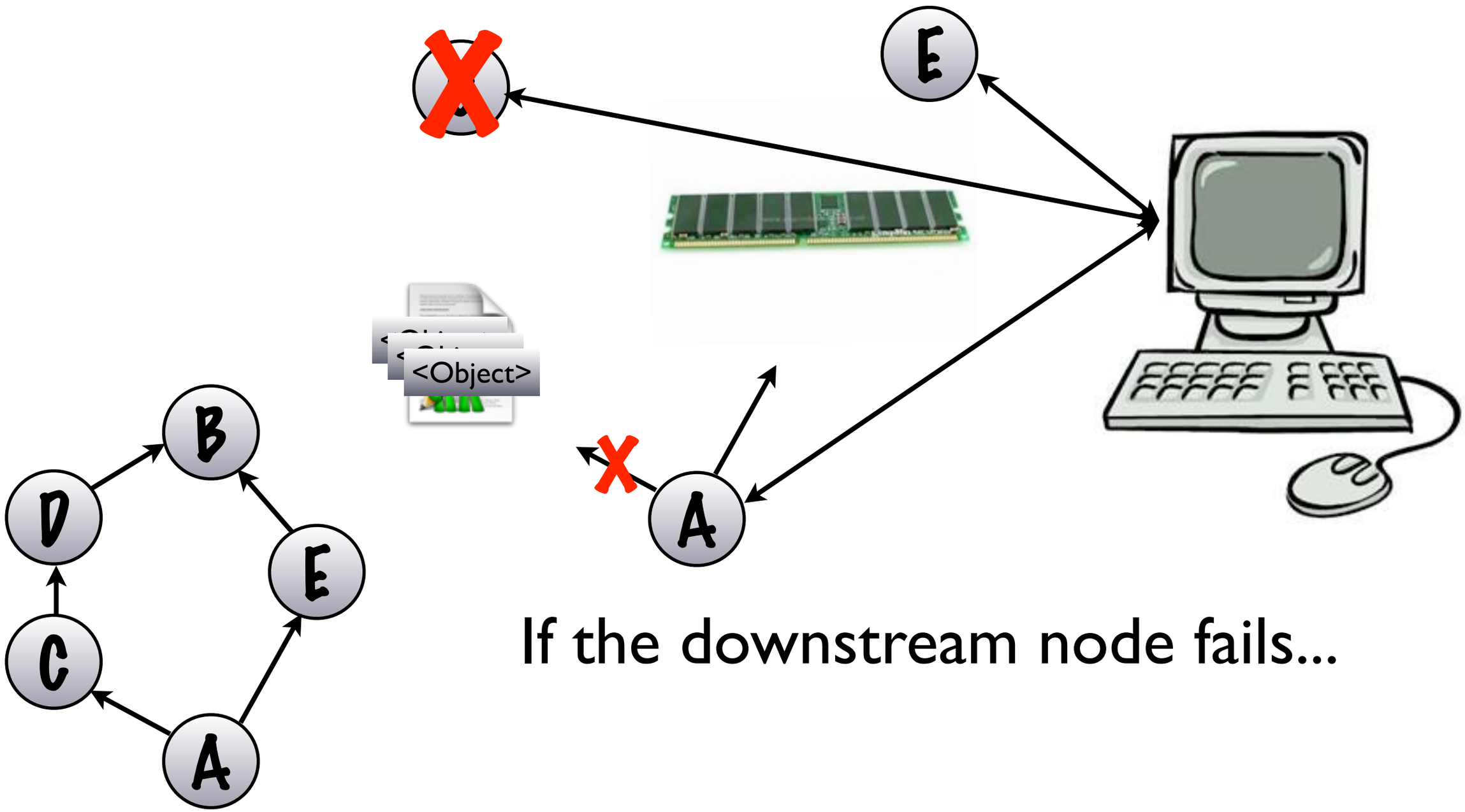
... and the downstream node is spawned.

Job Execution



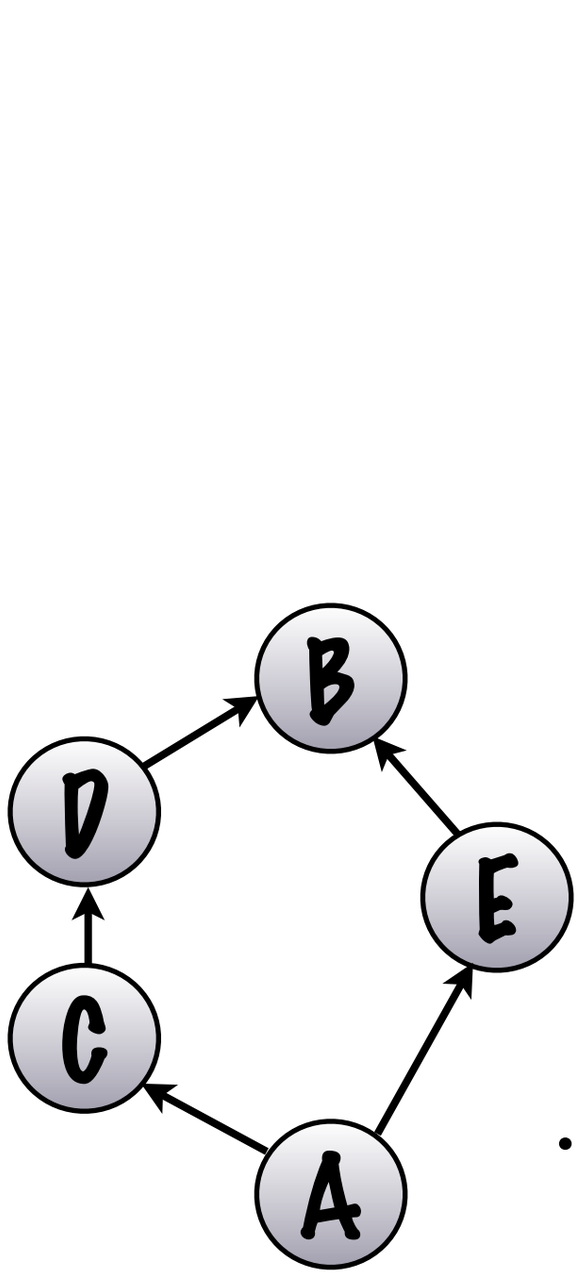
... and the downstream node is spawned.

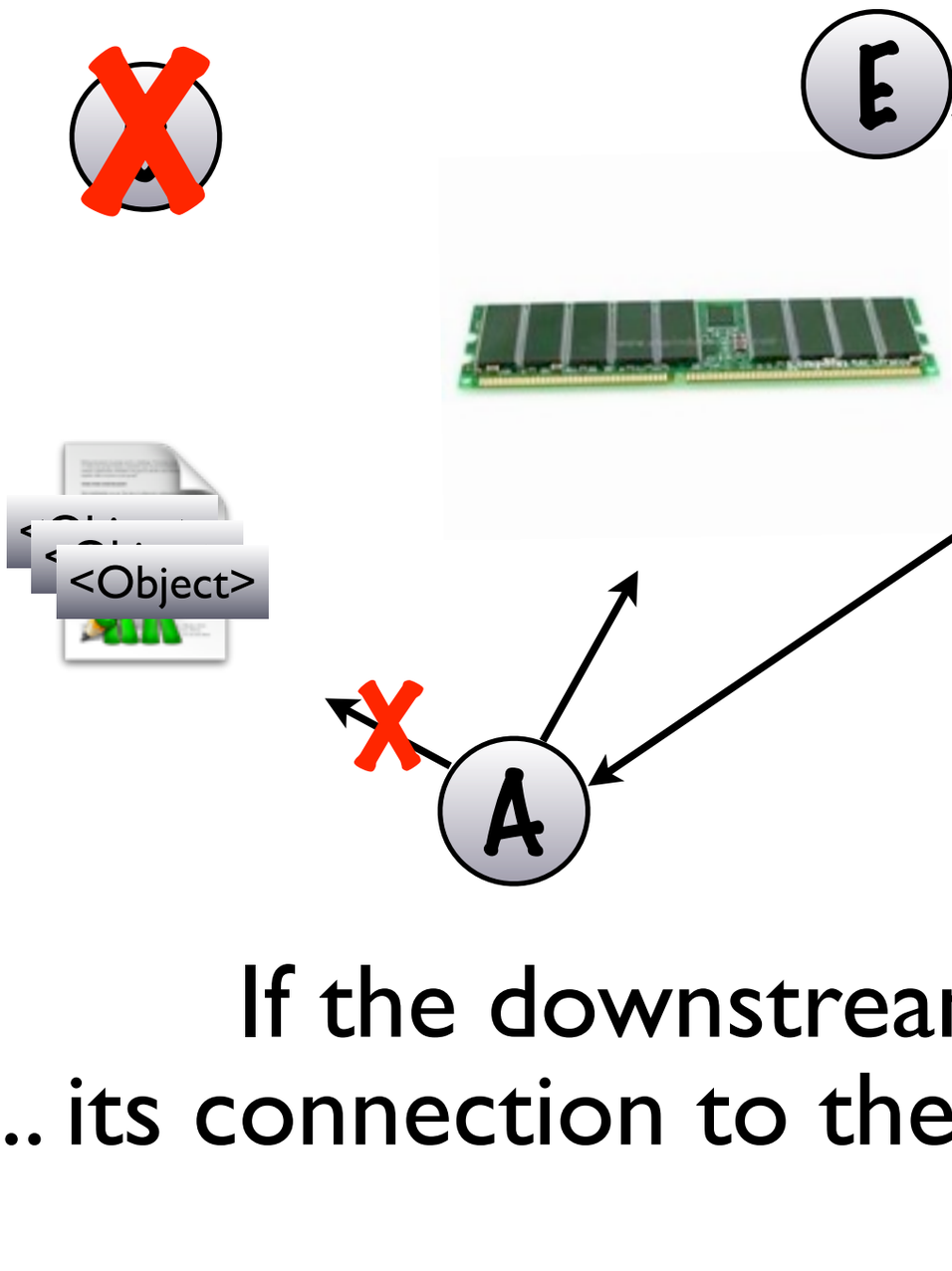
Job Execution

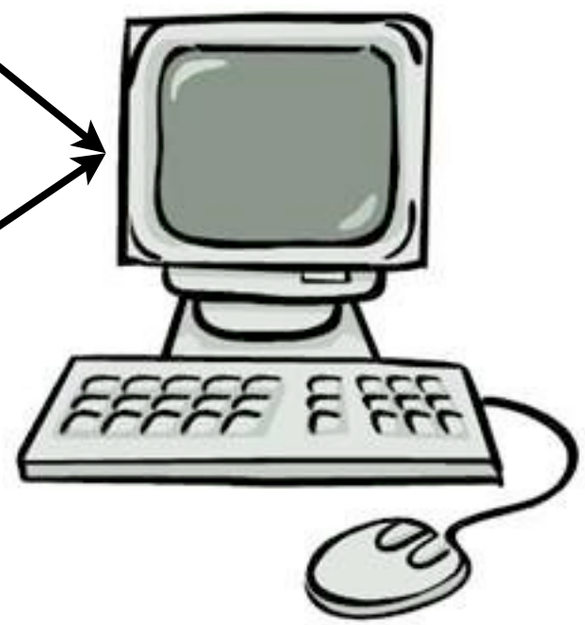


If the downstream node fails...

Job Execution

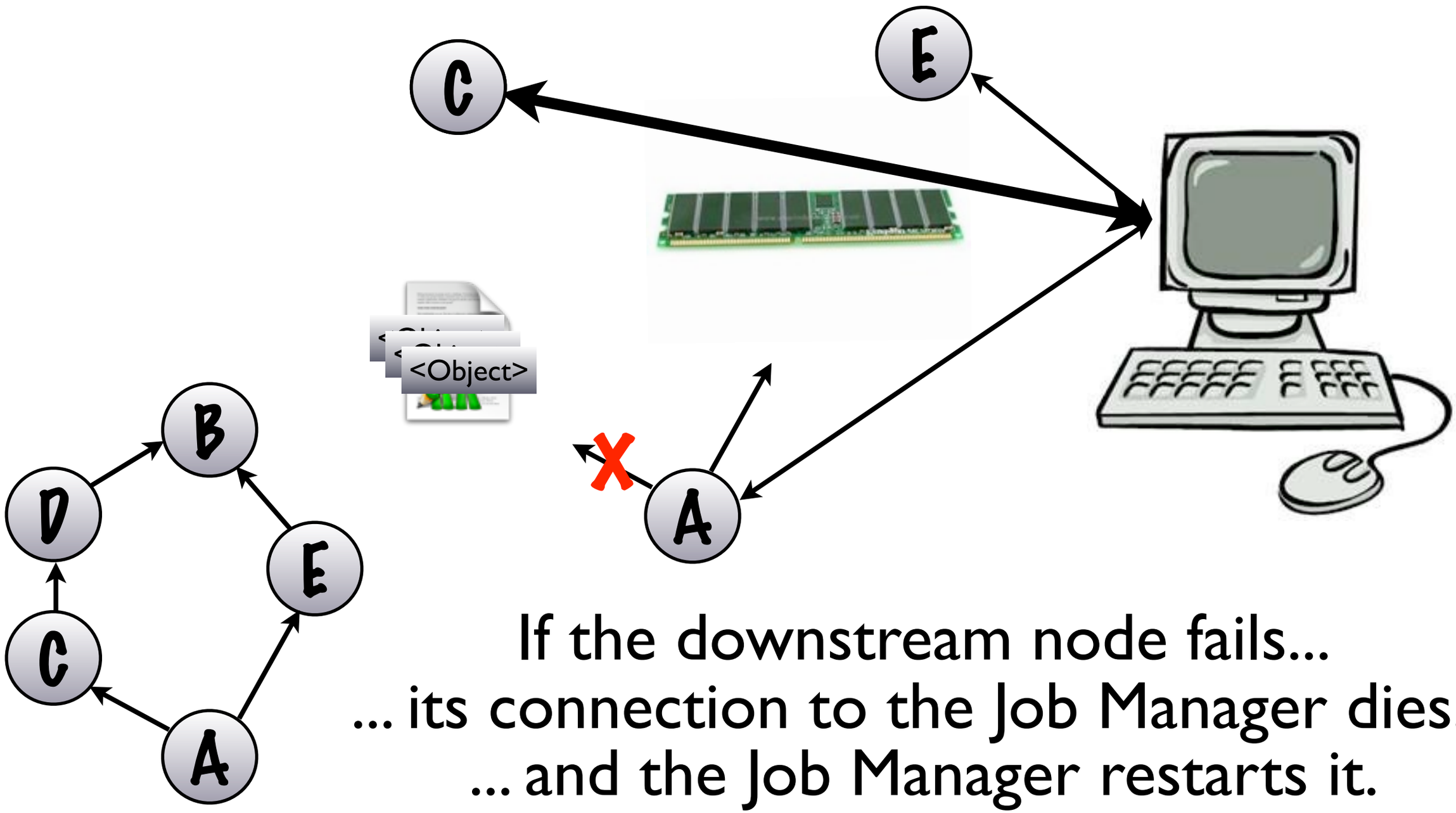






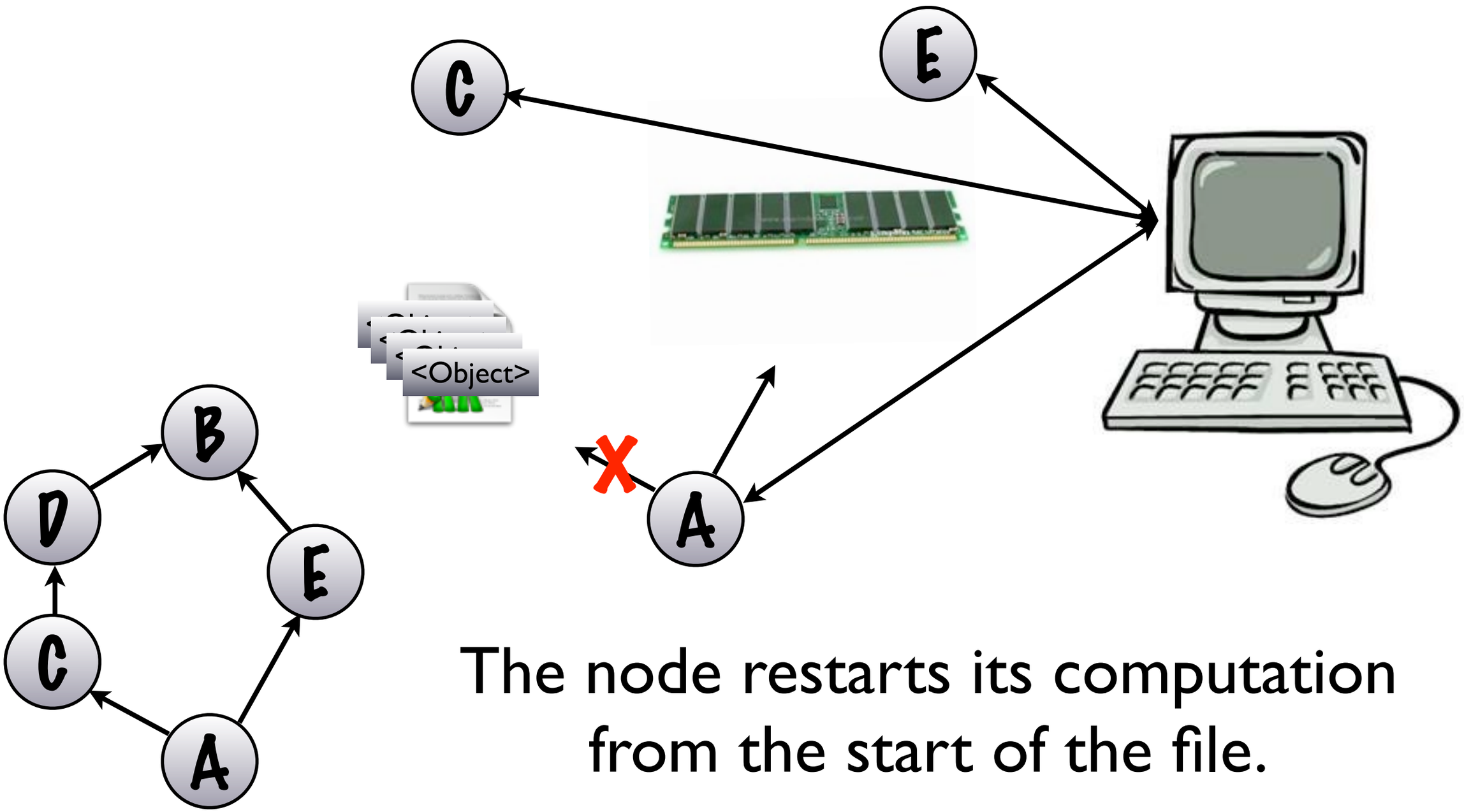
If the downstream node fails...
... its connection to the Job Manager dies...

Job Execution



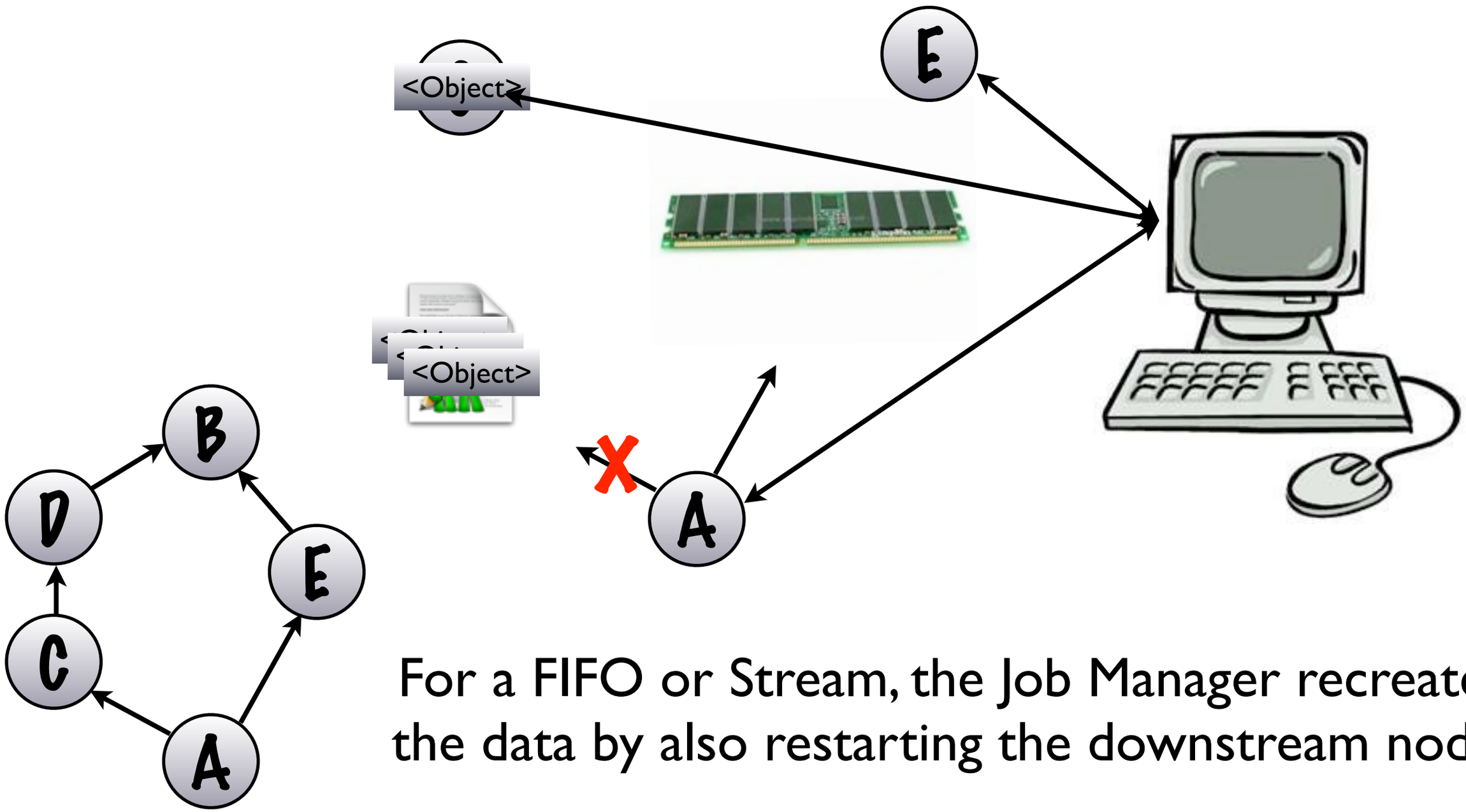
If the downstream node fails...
... its connection to the Job Manager dies...
... and the Job Manager restarts it.

Job Execution



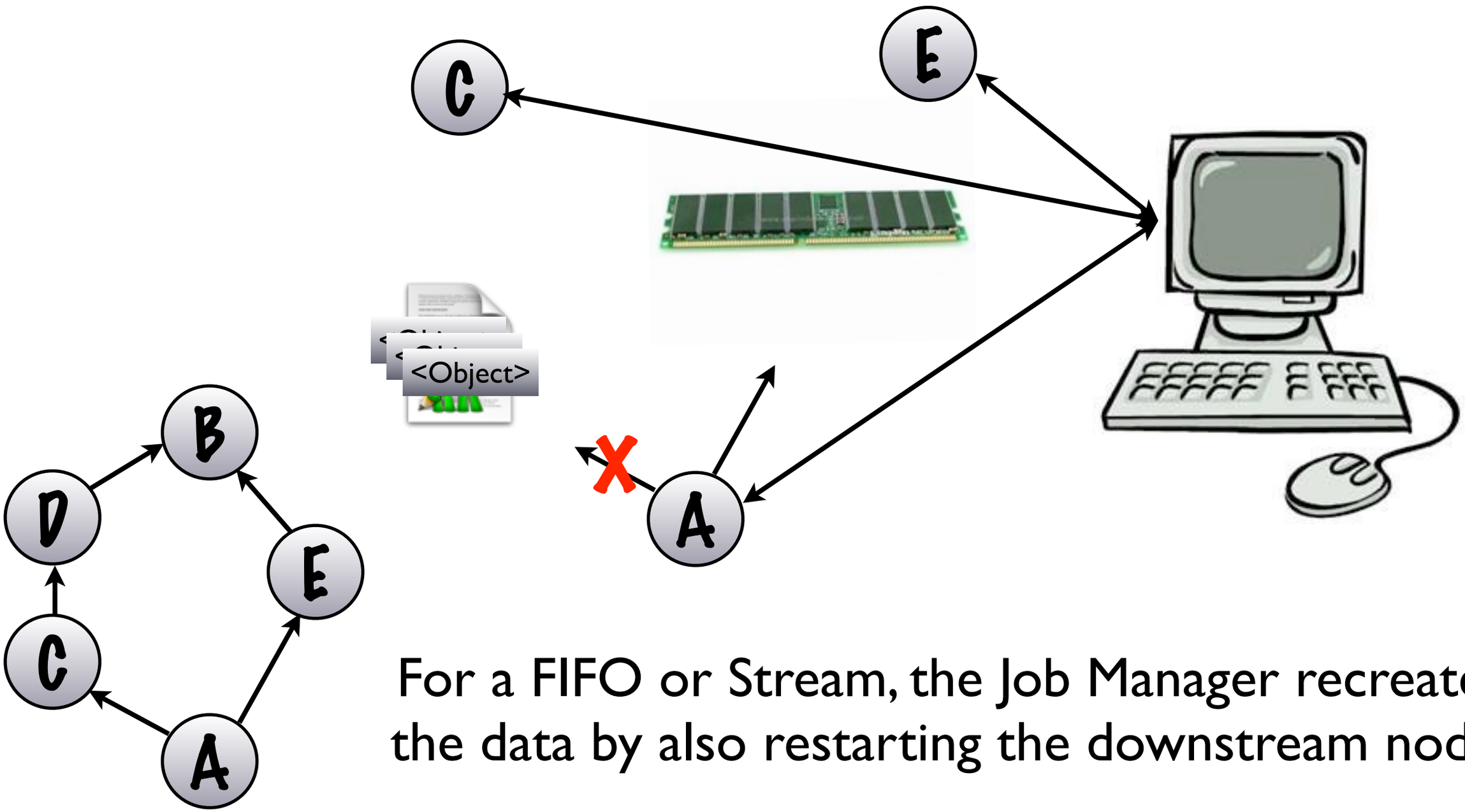
The node restarts its computation from the start of the file.

Job Execution



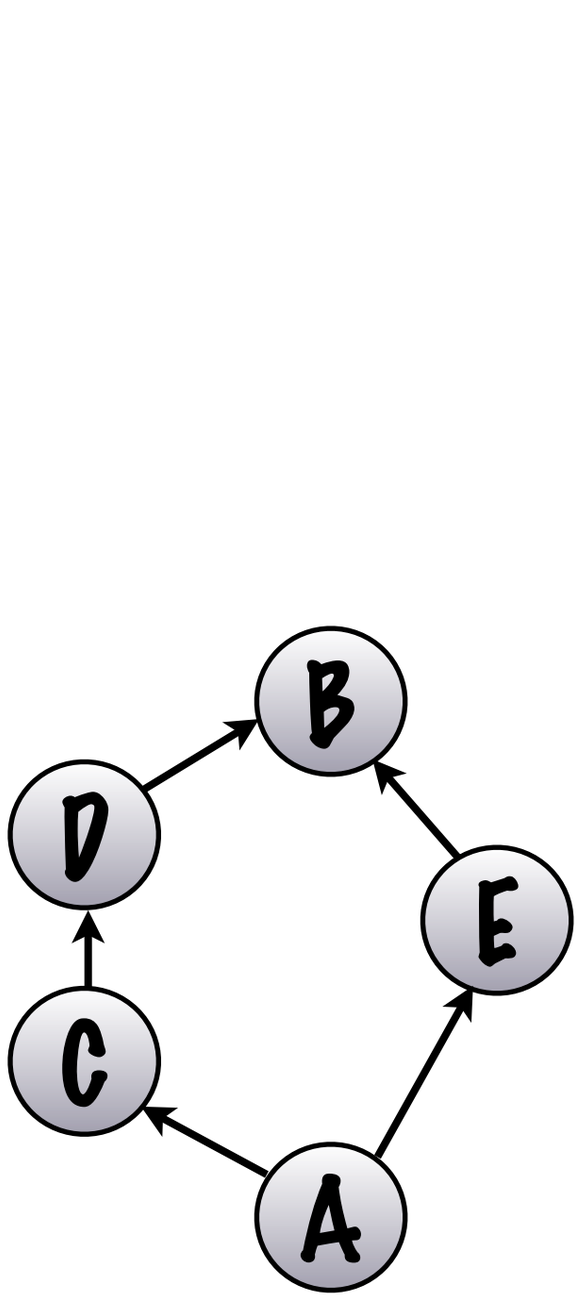
For a FIFO or Stream, the Job Manager recreates the data by also restarting the downstream node.

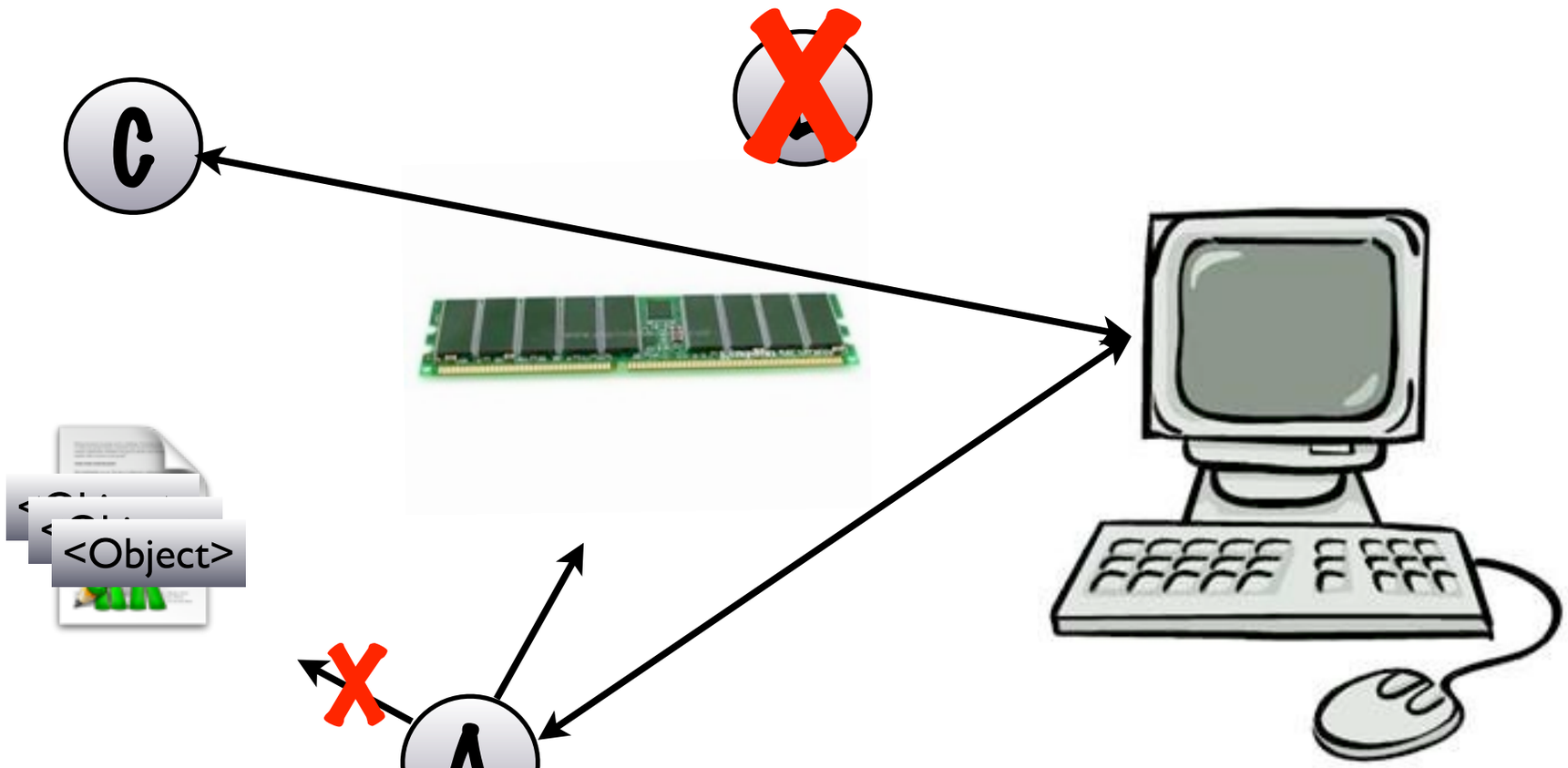
Job Execution



For a FIFO or Stream, the Job Manager recreates the data by also restarting the downstream node.

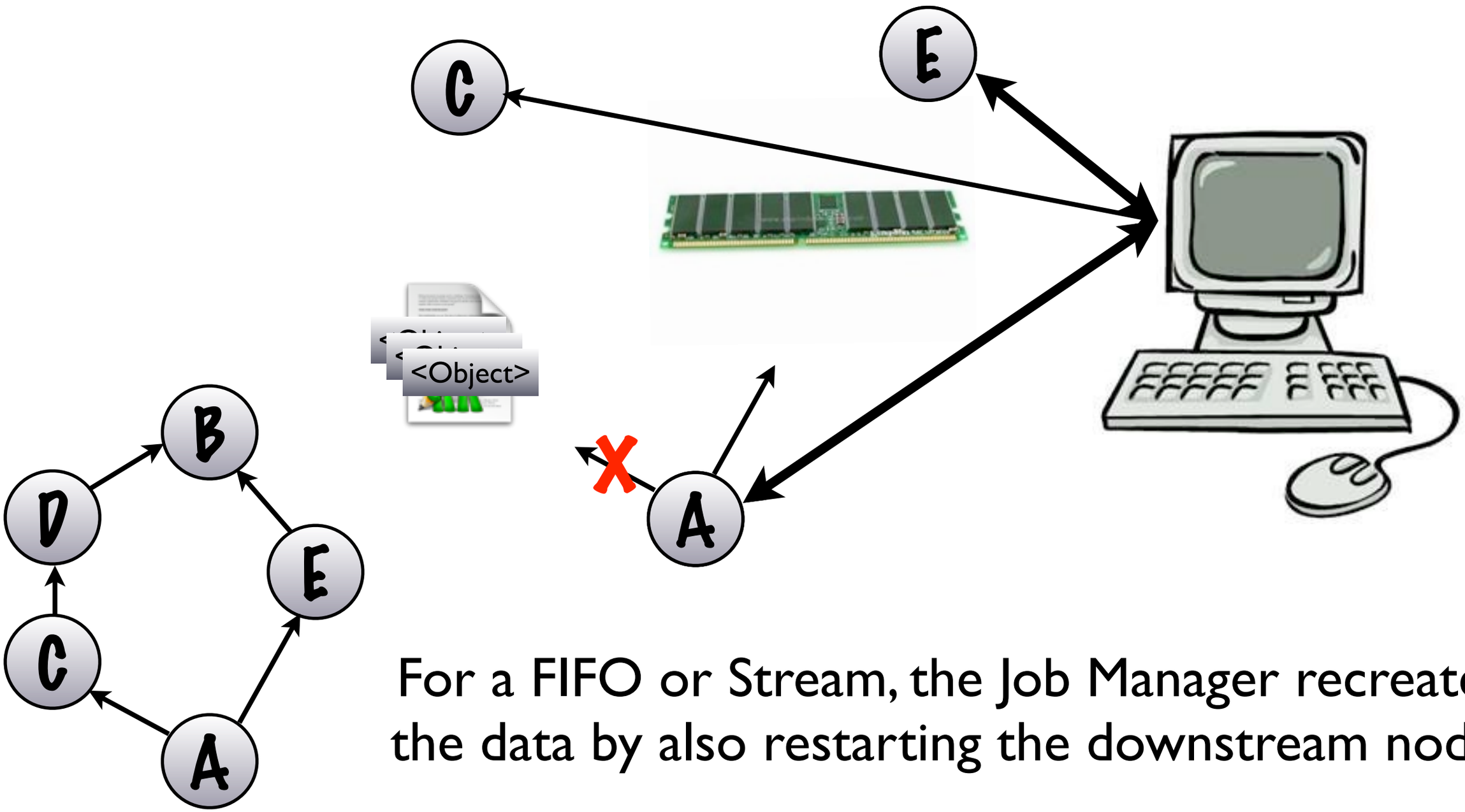
Job Execution





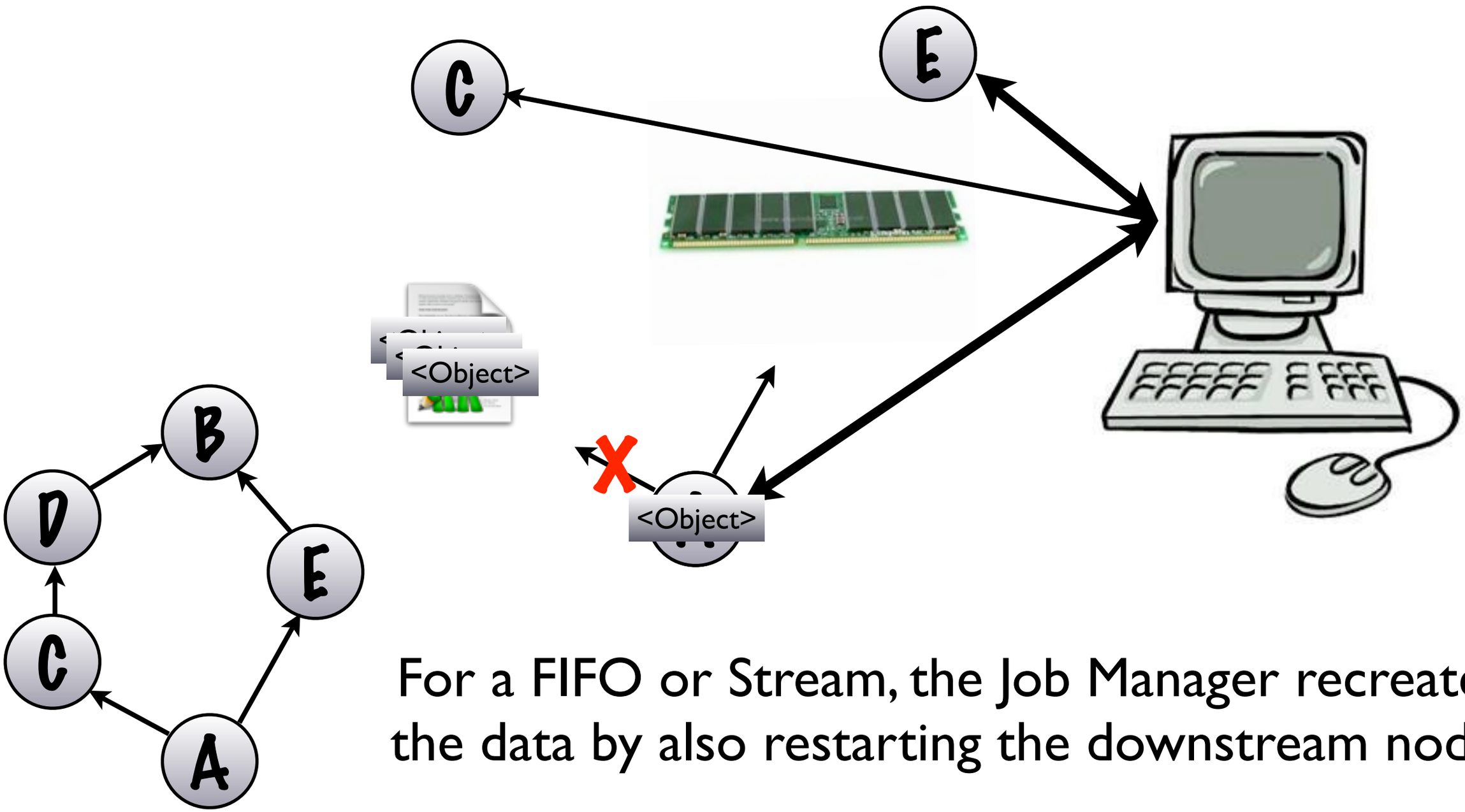
For a FIFO or Stream, the Job Manager recreates the data by also restarting the downstream node.

Job Execution



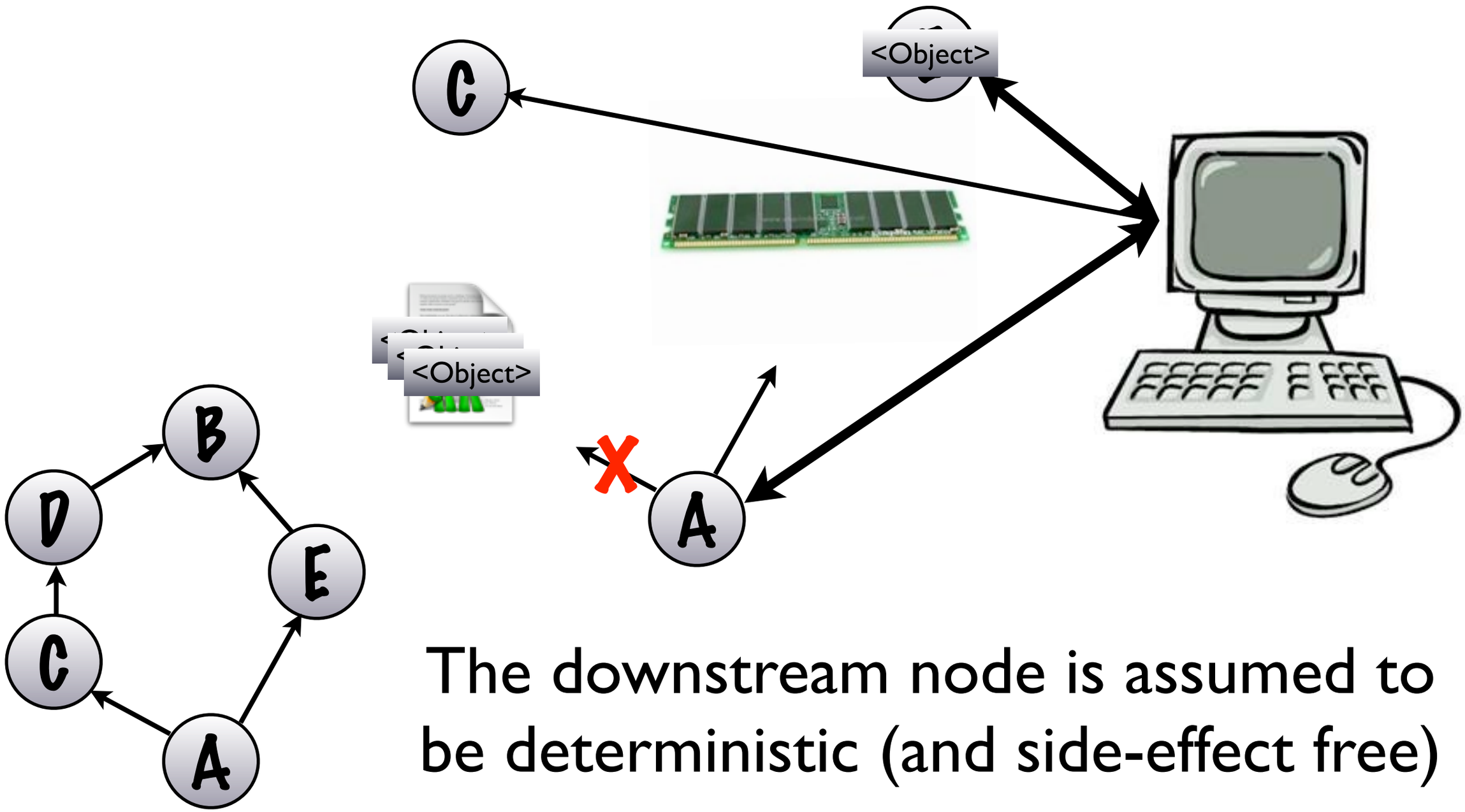
For a FIFO or Stream, the Job Manager recreates the data by also restarting the downstream node.

Job Execution



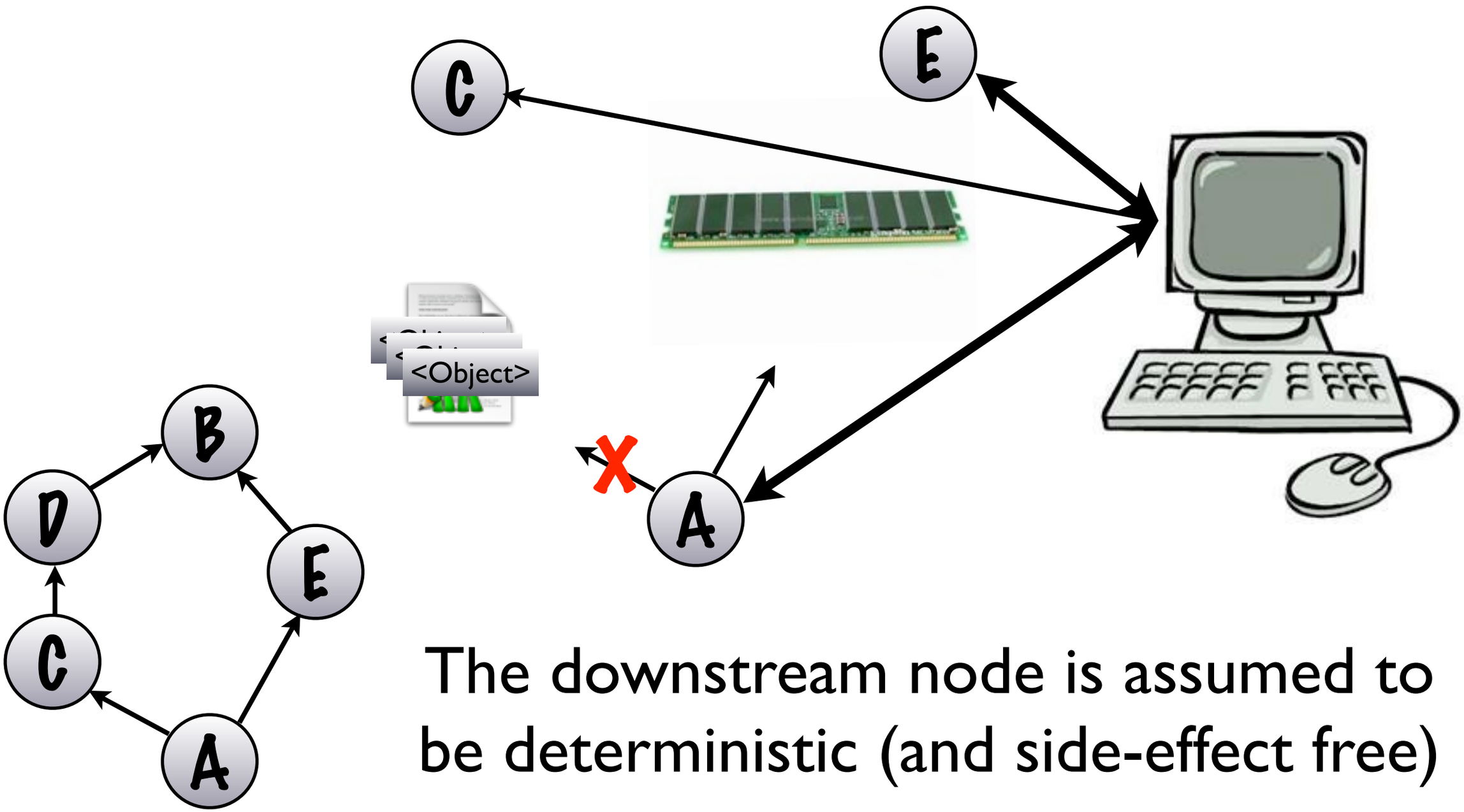
For a FIFO or Stream, the Job Manager recreates the data by also restarting the downstream node.

Job Execution



The downstream node is assumed to be deterministic (and side-effect free)

Job Execution

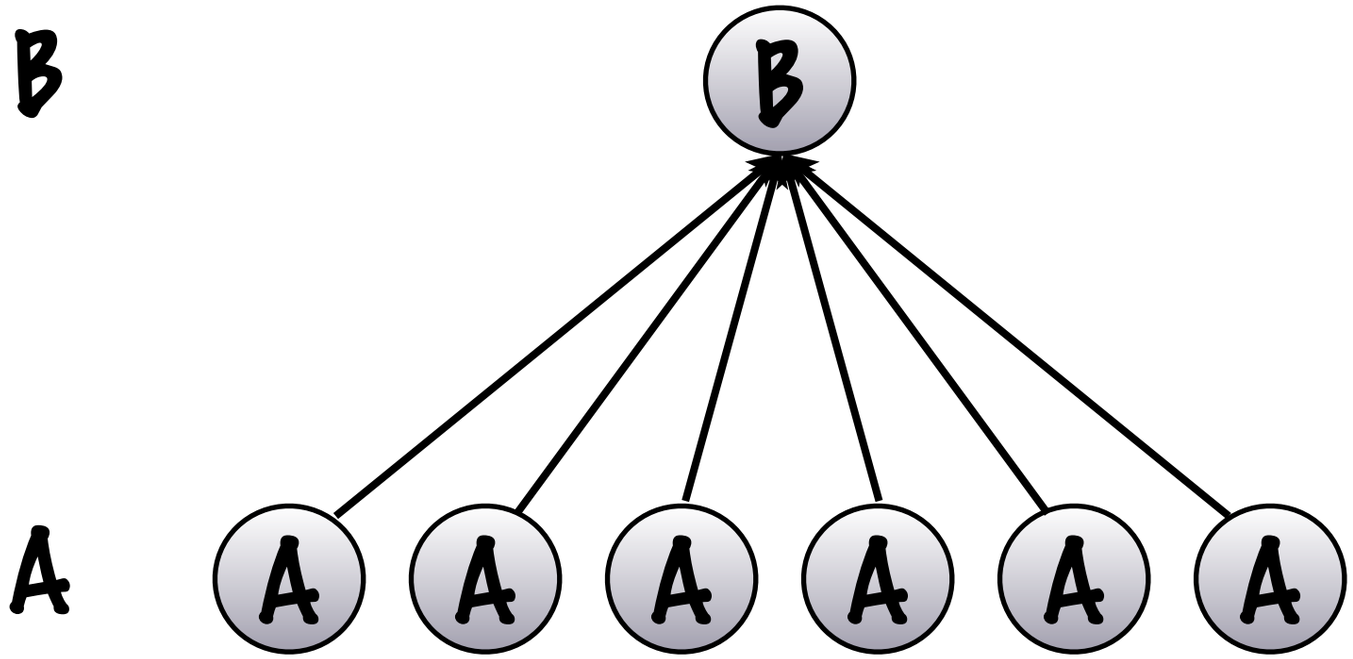


The downstream node is assumed to be deterministic (and side-effect free)

Runtime Graph Refinement

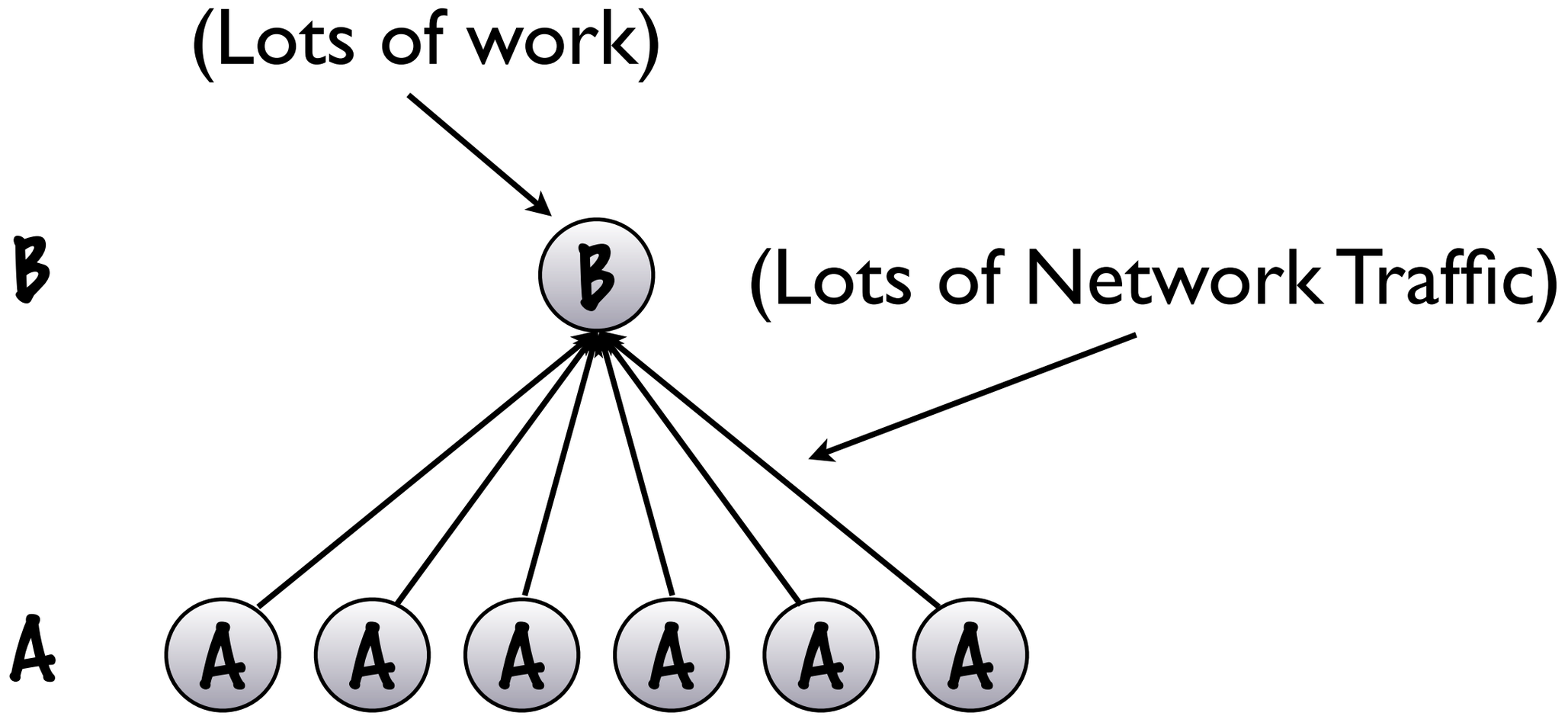
What about aggregation?

Runtime Graph Refinement



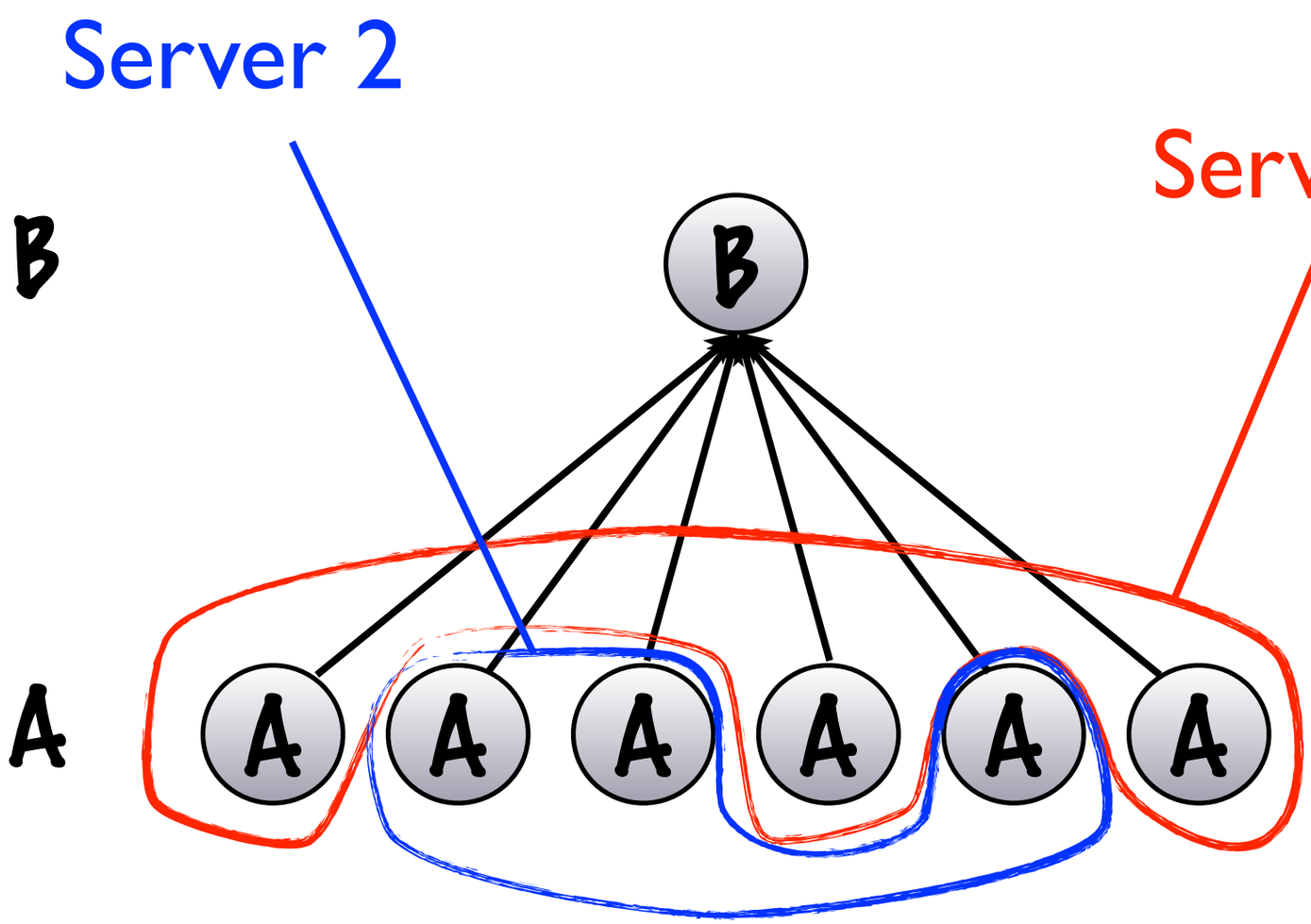
Aggregation is Expensive!

Runtime Graph Refinement



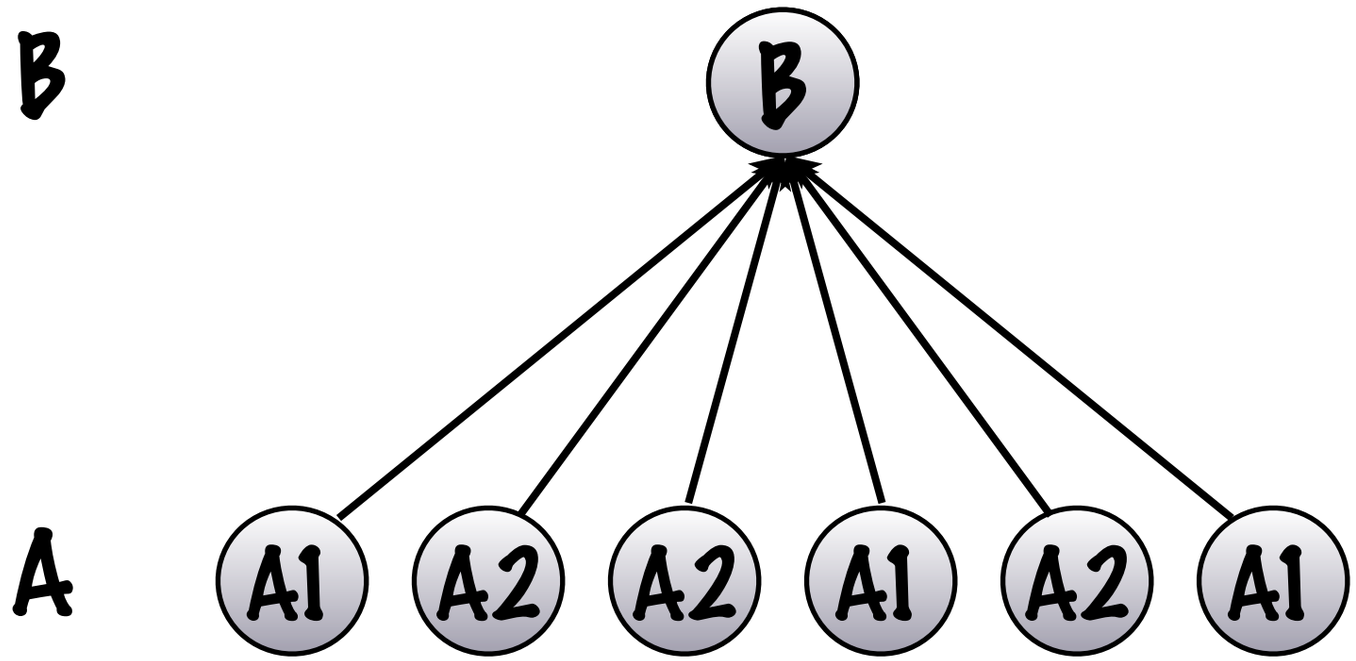
Aggregation is Expensive!

Runtime Graph Refinement



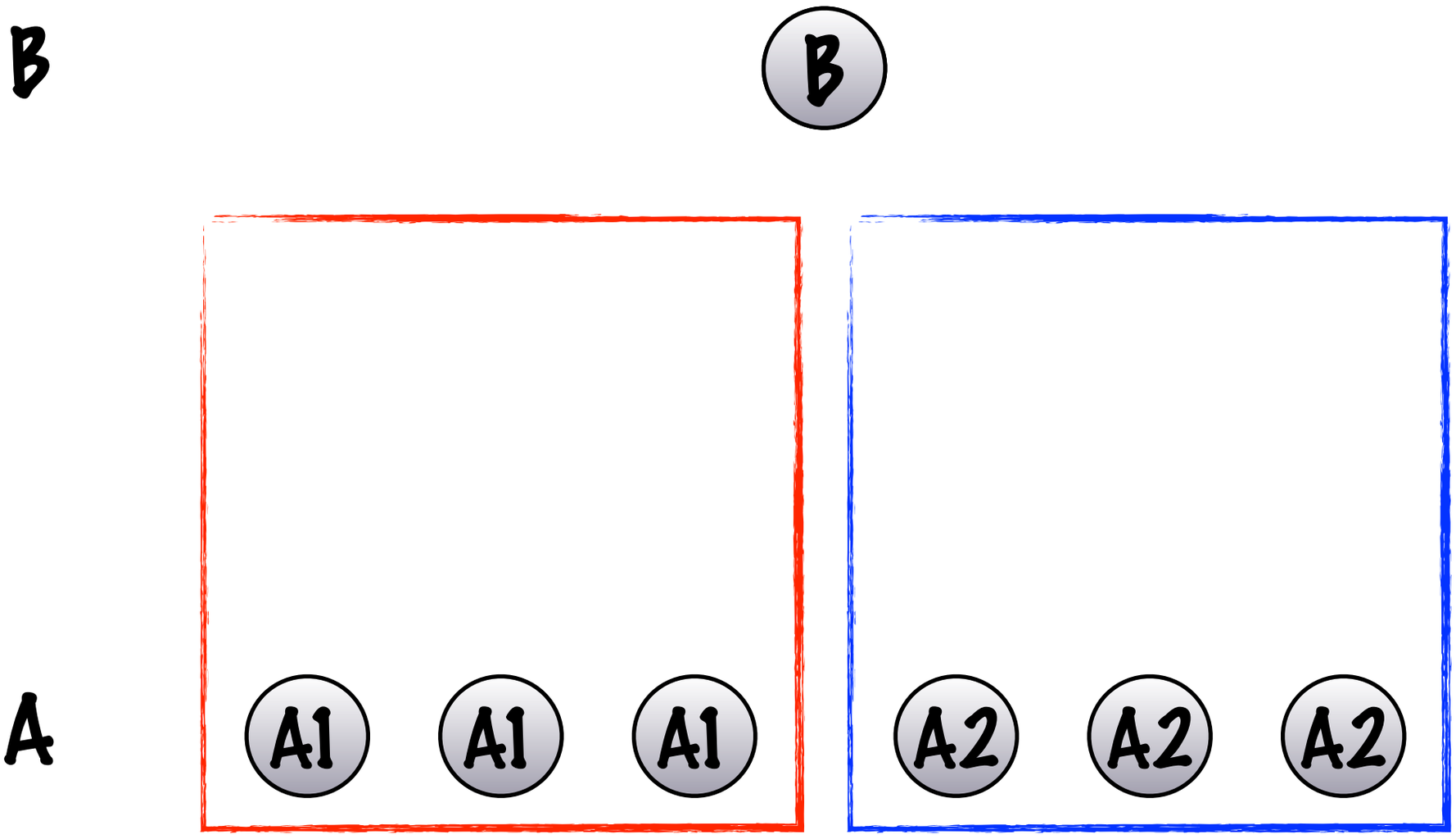
We can use runtime information to optimize!

Runtime Graph Refinement



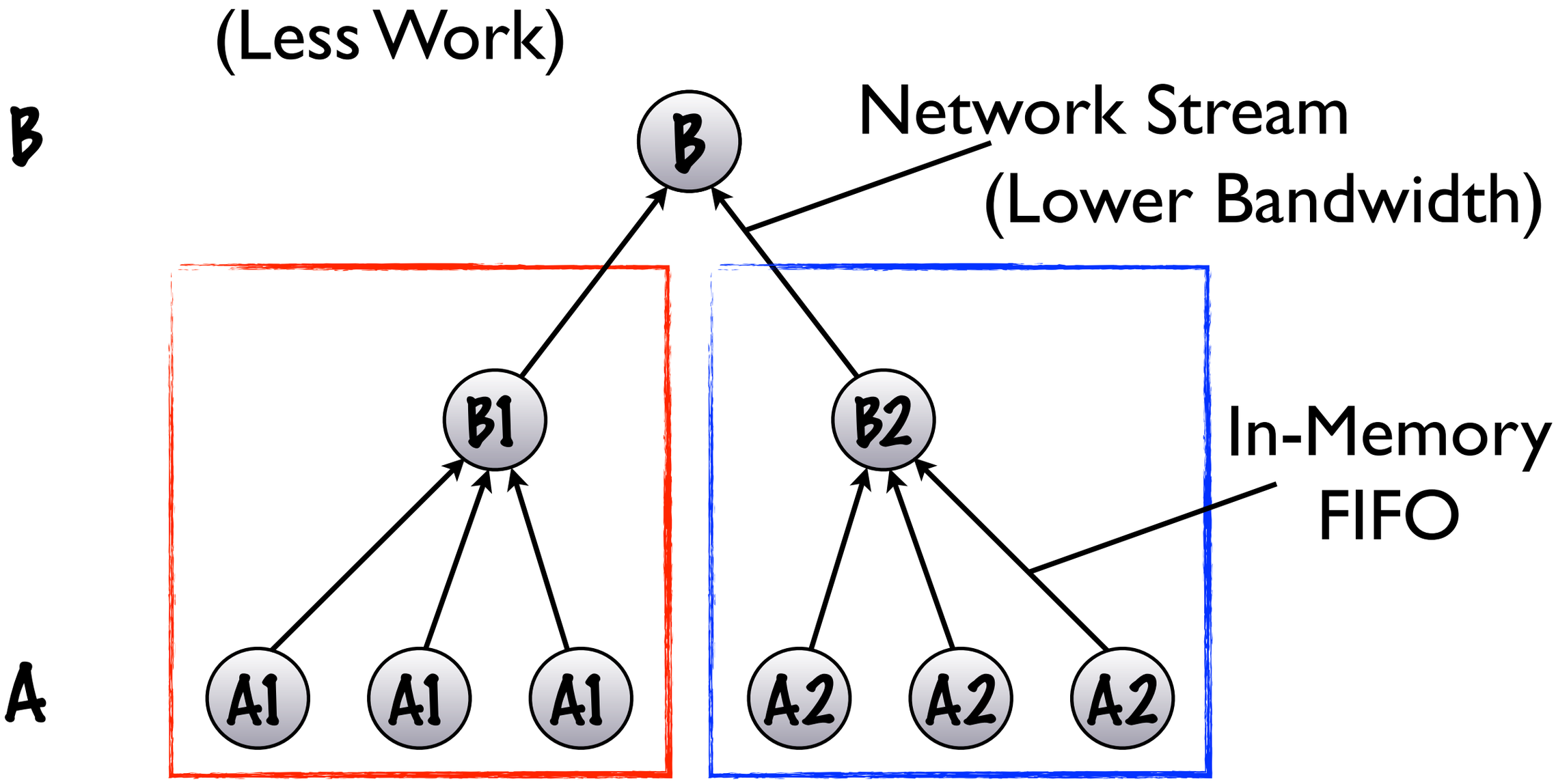
Once the Job Manager assigns nodes to machines...

Runtime Graph Refinement



... we can apply a **user-provided** function to pre-aggregate.

Runtime Graph Refinement



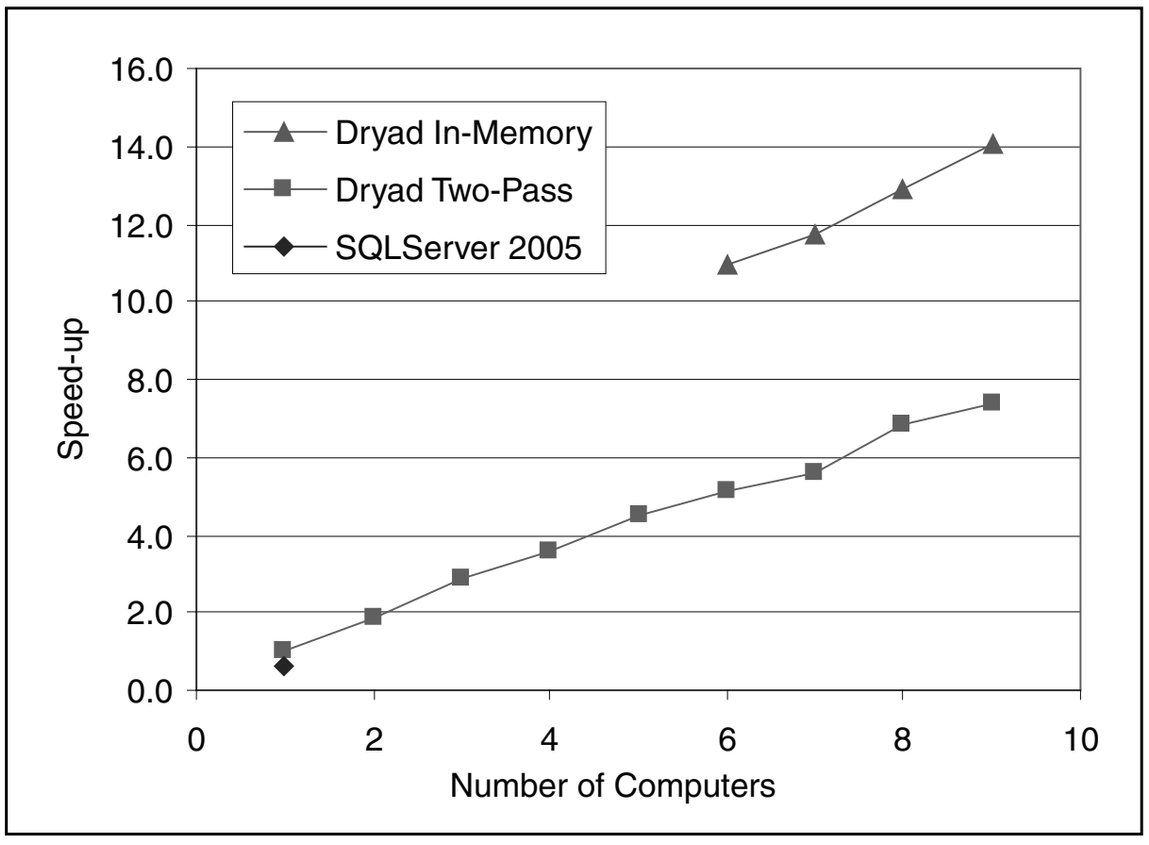
... we can apply a **user-provided** function to pre-aggregate.

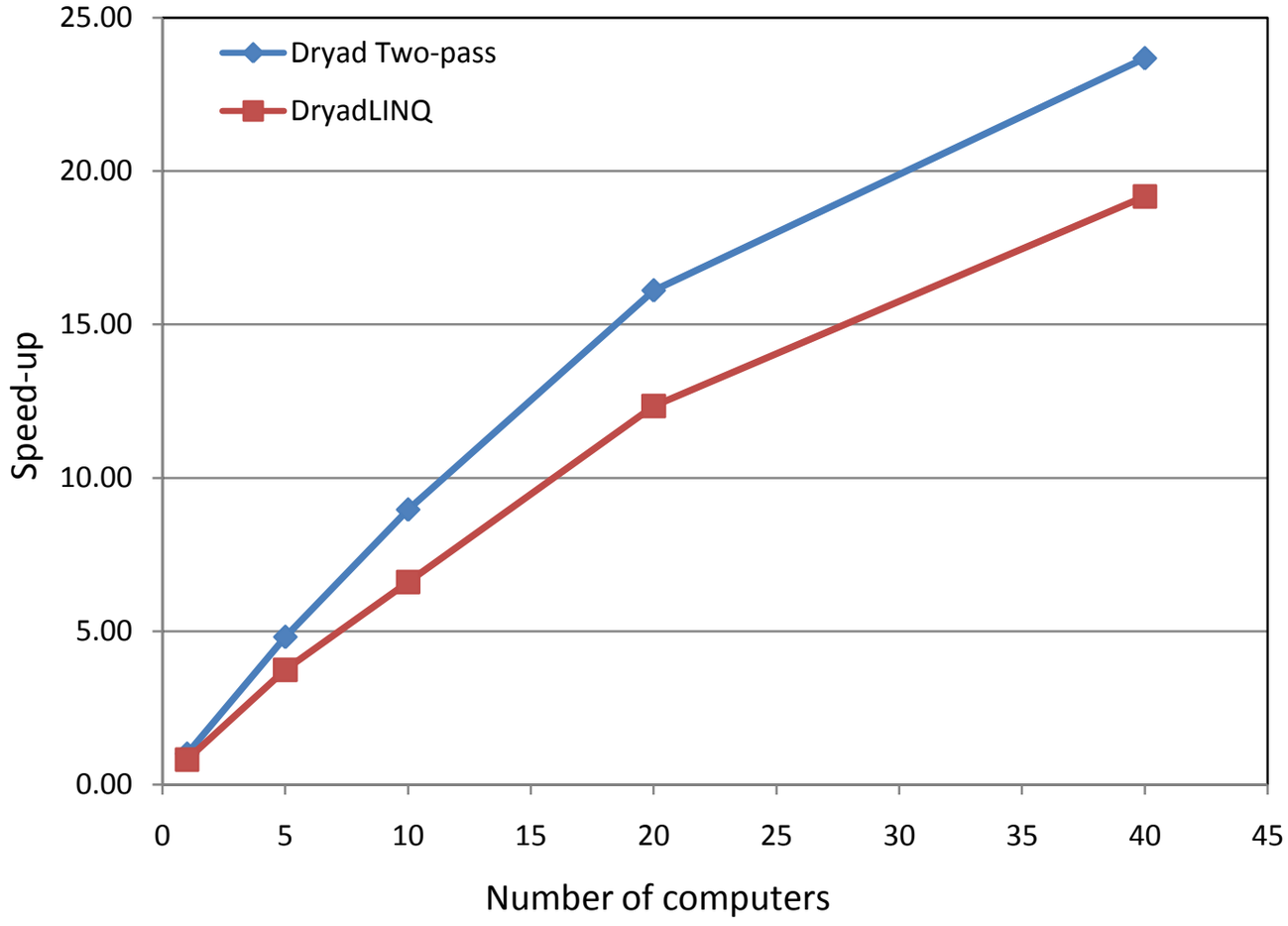
- Graph Programming
- Execution Model
- **Evaluation**

Evaluation

```
SELECT distinct p.objID
FROM   photoObjAll p
  JOIN neighbors n
    ON p.objID = n.objID
   AND n.objID < n.neighborObjID
  JOIN photoObjAll l
    ON l.objID = n.neighborObjID
   AND SimilarColor(l.rgb,n.rgb)
```

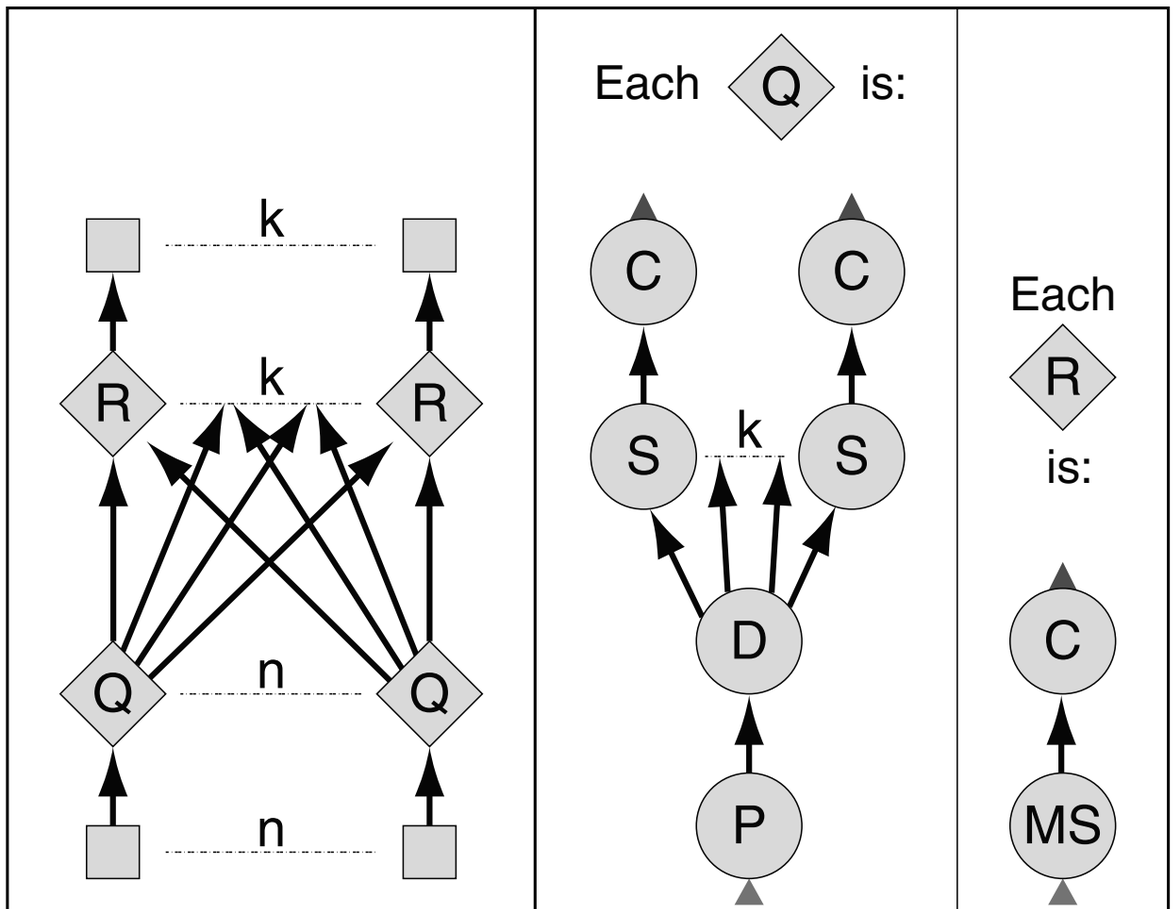
Evaluation


Dryad In-Memory
Dryad Two-Pass
SQLServer 2005
Speed-up
0.0
2.0
4.0
6.0
8.0
10.0
12.0
14.0
16.0
0
2
4
6
8
10
Number of Computers


Dryad Two-pass
DryadLINQ
Speed-up
0.00
5.00
10.00
15.00
20.00
25.00
0
5
10
15
20
25
30
35
40
45
Number of computers

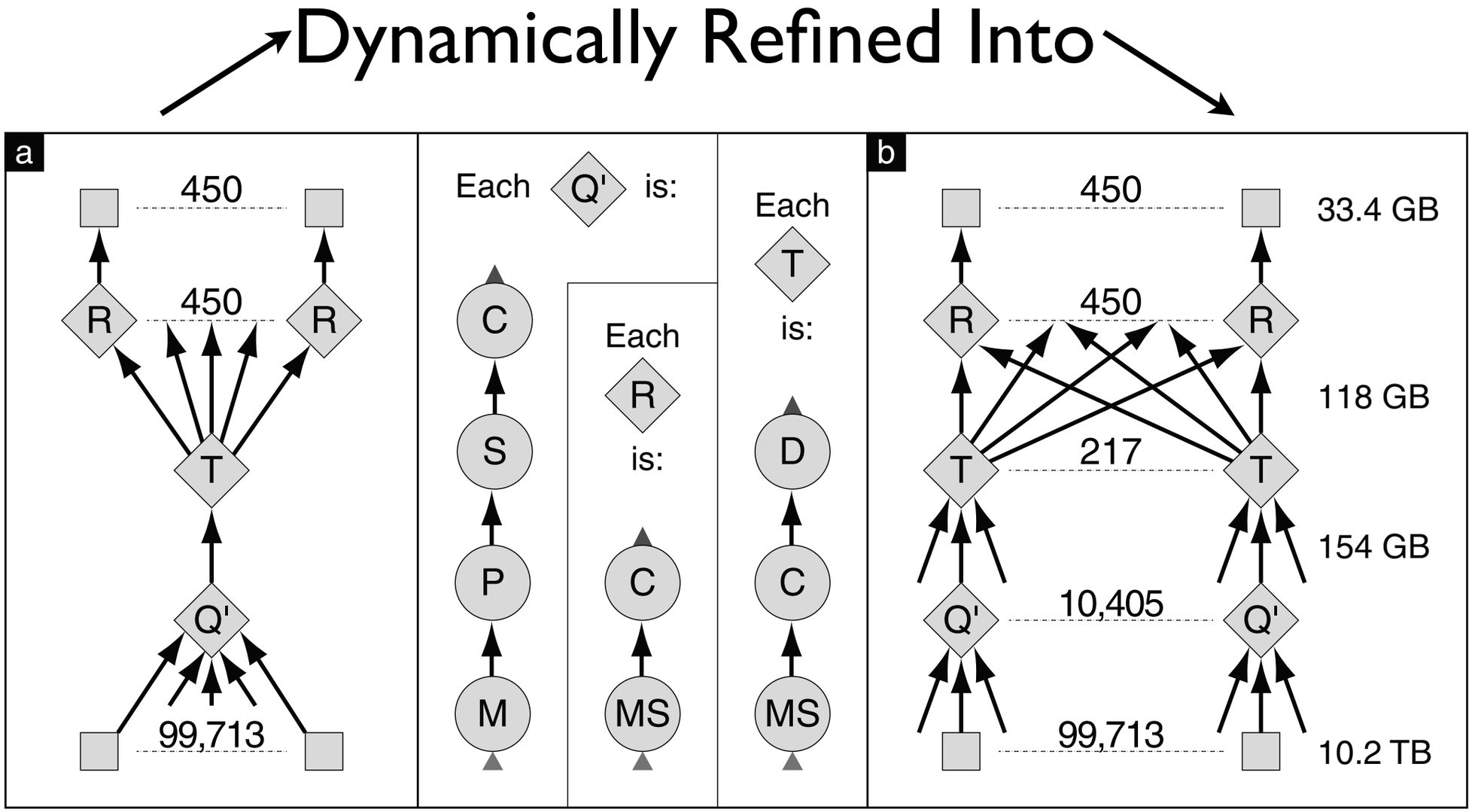
Evaluation

P: Read/Parse Input
D: Distribute Copies
S: In-Memory Sort
MS: Merge Sort
C: Count



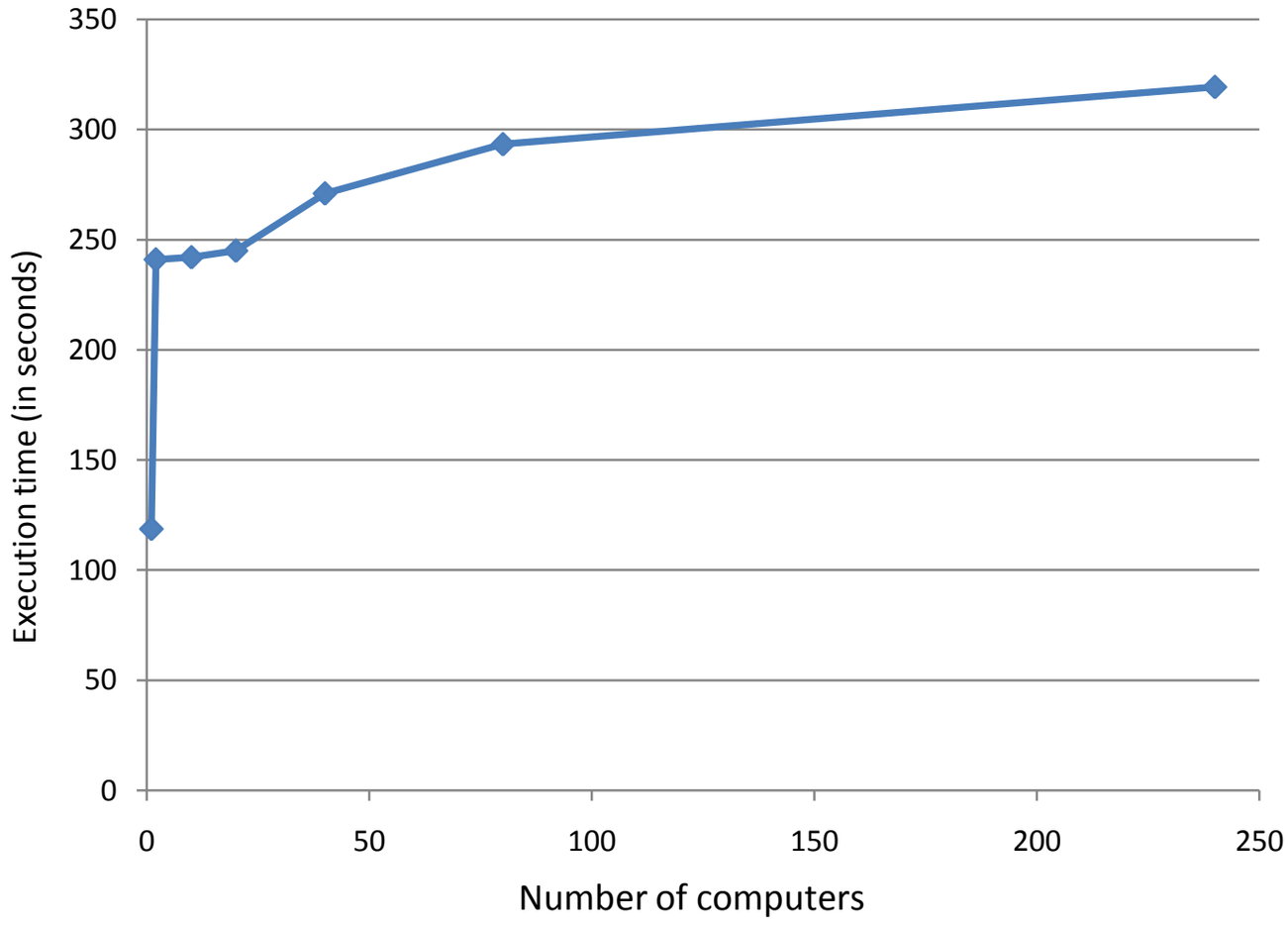
Time: ?? (really bad)

Evaluation



Time: 11 min 30 sec

Evaluation


350
300
250
200
150
100
50
0
0
50
100
150
200
250
Execution time (in seconds)
Number of computers

Conclusions

- The hard part of distributed programming is infrastructure.
- Programmers like thinking in code nuggets.
- Graphs are a natural representation of distributed computations.
- Dryad isn't used much, but production dataflow systems use virtually identical techniques.
  - E.g., Storm (Twitter's analytics infrastructure)

nathanmarz/storm · GitHub

GitHub, Inc. github.com/nathanmarz/storm

Reader

Bills School Languages Comics Info Projects Gaming Personal TV Tools Macros

nathanmarz/storm · GitHub

github

Signup and Pricing Explore GitHub Features Blog Sign in

PUBLIC nathanmarz / storm

Star 3,980 Fork 384

Code Network Pull Requests 10 Issues 78 Wiki Graphs

Distributed and fault-tolerant realtime computation: stream processing, continuous computation, distributed RPC, and more — Read more

http://storm-project.net

Clone in Mac ZIP HTTP Git Read-Only https://github.com/nathanmarz/storm.git Read-Only access

branch: master Files Commits Branches 33 Tags Downloads 17

Latest commit to the master branch

update changelog

nathanmarz authored 3 days ago commit 24ae774832

storm /

name	age	message	history
bin	8 days ago	Fix directory bug in install_zmq.sh [minghan]	
conf	4 days ago	Implemented pluggable spout wait strategies [nathanmarz]	
log4j	7 months ago	add storm dir for logs folder to use absolute path for logs folder [haitaoyao]	
src	4 days ago	added ISchemableSpout interface [nathanmarz]	
test	5 days ago	formatting [nathanmarz]	
.gitignore	a month ago	Adding IntelliJ files to ignore [sjoerdmulder]	
CHANGELOG.md	3 days ago	update changelog [nathanmarz]	
LICENSE.html	a year ago	release commit [nathanmarz]	

Conclusions

- The hard part of distributed programming is infrastructure.
- Programmers like thinking in code nuggets.
- Graphs are a natural representation of distributed computations.
- Dryad isn't used much, but production dataflow systems use virtually identical techniques.
  - E.g., Storm (Twitter's analytics infrastructure)

Conclusions

- The hard part of distributed programming is infrastructure.
- Programmers like thinking in code nuggets.
- Graphs are a natural representation of distributed computations.
- Dryad isn't used much, but production dataflow systems use virtually identical techniques.
  - E.g., Storm (Twitter's analytics infrastructure)

Questions?