

RPC Wrapup and Distributed File Systems

Amin Vahdat

CSE 123b

May 16, 2006

Annoucements

- Second assignment was due May 15
- Third assignment available by Thurs

Remote Procedure Call

XML-RPC

- XML is a standard for describing structured documents
 - Uses tags to define structure: **<tag> ... </tag>** demarcates an element
 - Tags have no predefined semantics ...
 - ... except when document refers to a specific namespace
 - Elements can have attributes, which are encoded as name-value pairs

A well-formed XML document corresponds to an element tree

- **<?xml version="1.0"?>**
<methodCall>
 <methodName>SumAndDifference</methodName>
 <params>
 <param><value><i4>40</i4></value></param>
 <param><value><i4>10</i4></value></param>
 </params>
</methodCall>

XML-RPC Wire Format

- Scalar values
 - Represented by a **<value><type> ... </type></value>** block
- Integer
 - `<i4>12</i4>`
- Boolean
 - `<boolean>0</boolean>`
- String
 - `<string>Hello world</string>`
- Double
 - `<double>11.4368</double>`
- Also Base64 (binary), DateTime, etc.

XML-RPC Wire Format (struct)

- Structures
 - Represented as a set of **<member>**s
 - Each member contains a **<name>** and a **<value>**
- **<struct>**
 - <member>**
 - <name>lowerBound</name>**
 - <value><i4>18</i4></value>**
 - </member>**
 - <member>**
 - <name>upperBound</name>**
 - <value><i4>139</i4></value>**
 - </member>**
- </struct>**

XML-RPC Wire Format (Arrays)

- Arrays

- A single **<data>** element, which
- contains any number of **<value>** elements

- **<array>**

<data>

<value><i4>12</i4></value>

<value><string>Egypt</string></value>

<value><boolean>0</boolean></value>

<value><i4>-31</i4></value>

</data>

</array>

XML-RPC Request

- HTTP POST message
 - URI interpreted in an implementation-specific fashion
 - Method name passed to the server program
- **POST /RPC2 HTTP/1.1**
Content-Type: text/xml
User-Agent: XML-RPC.NET
Content-Length: 278
Expect: 100-continue
Connection: Keep-Alive
Host: localhost:8080
<?xml version="1.0"?>
<methodCall>
<methodName>SumAndDifference</methodName>
<params>
<param><value><i4>40</i4></value></param>
<param><value><i4>10</i4></value></param>
</params>
</methodCall>

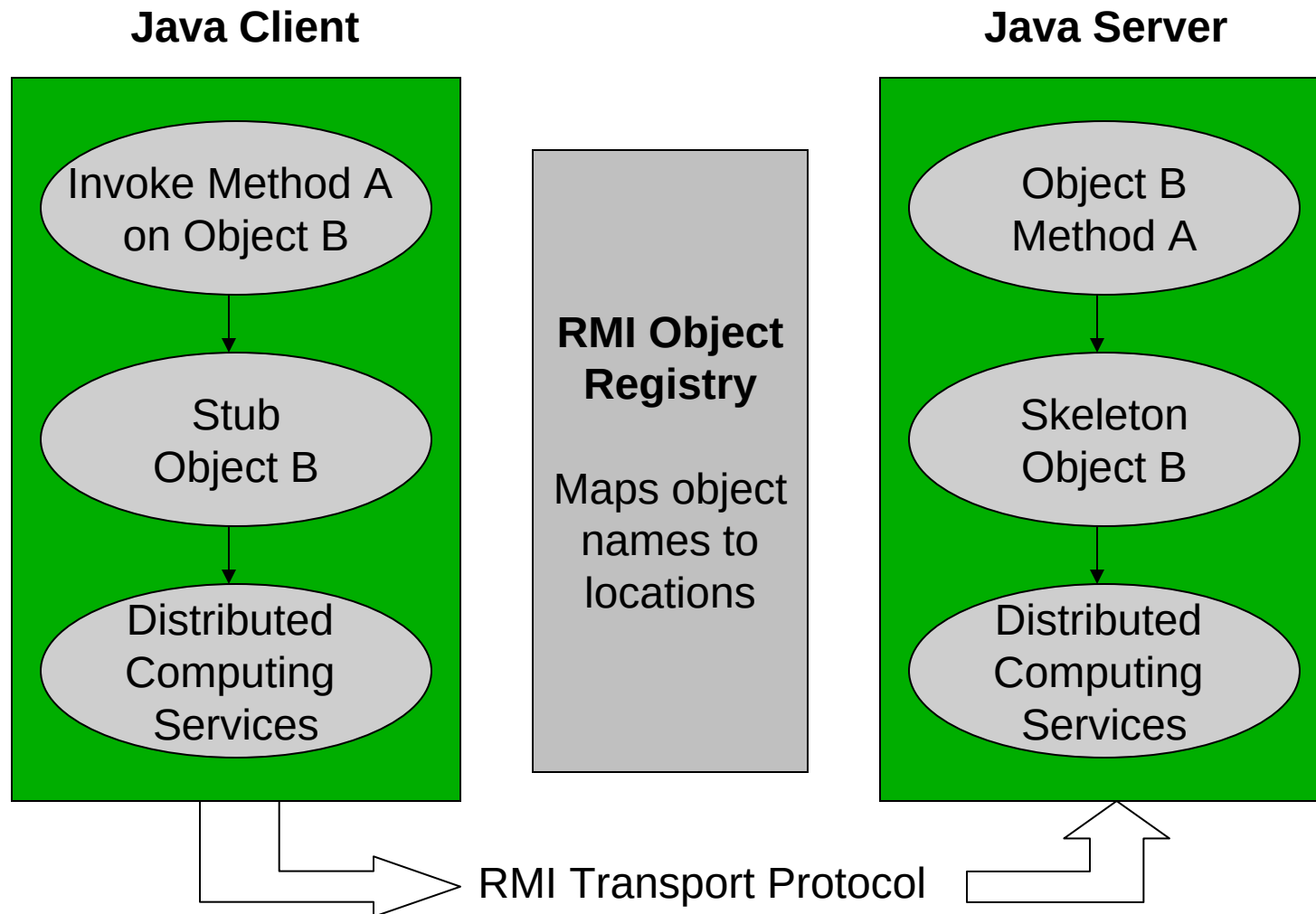
XML-RPC Response

- HTTP Response
 - Lower-level error returned as an HTTP error code
 - Application-level errors returned as a **<fault>** element (next slide)
- HTTP/1.1 200 OK
Date: Mon, 22 Sep 2003 21:52:34 GMT
Server: Microsoft-IIS/6.0
Content-Type: text/xml
Content-Length: 467
<?xml version="1.0"?>
<methodResponse>
<params><param>
<value><struct>
<member><name>sum</name><value><i4>50</i4></value></member>
<member><name>diff</name><value><i4>30</i4></value></member>
</struct></value>
</param></params>
</methodResponse>

XML-RPC Fault Handling

- Another kind of a MethodResponse
- ```
<?xml version="1.0"?>
<methodResponse>
 <fault>
 <value><struct>
 <member>
 <name>faultCode</name>
 <value><i4>500</i4></value>
 </member>
 <member>
 <name>faultString</name>
 <value><string>Arg `a' out of range</string></value>
 </member>
 </struct></value>
 </fault>
</methodResponse>
```

# RMI Architecture



(thanks to David Del Vecchio)

# RMI Example (Server)

## **Hello.java**

---

```
import java.rmi.*;

public interface HelloInterface extends Remote {
 public String say() throws RemoteException;
}
```

## **HelloInterface.java**

---

```
import java.rmi.*; import java.rmi.server.*;
public class Hello extends UnicastRemoteObject implements HelloInterface {
 private String message;

 public Hello (String msg) throws RemoteException { message = msg; }

 public String say() throws RemoteException {
 return message;
 }

 public static void main (String[] args) {
 System.setSecurityManager(new RMISecurityManager());
 try {
 Naming.bind ("Hello", new Hello ("Hello, world!"));
 } catch (Exception e) { System.out.println ("Server failed"); }
 }
}
```

# RMI Example (Client)

## HelloClient.java

---

```
import java.rmi.*;

public class HelloClient {
 public static void main (String[] args) {
 System.setSecurityManager(new RMISecurityManager());
 try {
 HelloInterface hello = (HelloInterface)
 Naming.lookup ("//(hostname)/Hello");
 System.out.println (hello.say());
 } catch (Exception e) {
 System.out.println ("HelloClient exception: " + e);
 }
 }
}
```

- The rmic utility is used to generate stubs and skeletons
- Once started, server will stay active
- RMI daemon (rmid) can be used to create (activate) objects on the fly

# RAID

(thanks to David Patterson)

## Replace Small Number of Large Disks with Large Number of Small Disks! (1988 Disks)

	IBM 3390K	IBM 3.5" 0061	x70
Capacity	20 GBytes	320 MBytes	23 GBytes
Volume	97 cu. ft.	0.1 cu. ft.	11 cu. ft. <b>9X</b>
Power	3 KW	11 W	1 KW <b>3X</b>
Data Rate	15 MB/s	1.5 MB/s	120 MB/s <b>8X</b>
I/O Rate	600 I/Os/s	55 I/Os/s	3900 IOs/s <b>6X</b>
MTTF	250 KHrs	50 KHrs	??? Hrs
Cost	\$250K	\$2K	\$150K

Disk Arrays have potential for large data and I/O rates, high MB per cu. ft., high MB per KW, but what about reliability?

## Array Reliability

- **Reliability of N disks = Reliability of 1 Disk  $\div$  N**

50,000 Hours  $\div$  70 disks = 700 hours

Disk system MTTF: Drops from 6 years to 1 month!

- **Arrays (without redundancy) too unreliable to be useful!**

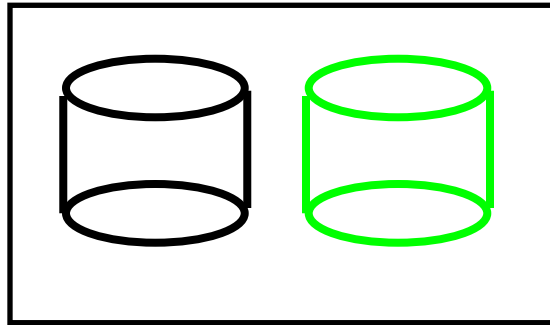
**Hot spares support reconstruction in parallel with access: very high media availability can be achieved**

# Redundant Arrays of (Inexpensive) Disks

- Files are "striped" across multiple disks
- Redundancy yields high data availability
  - Availability: service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array
  - Capacity penalty to store redundant info
  - Bandwidth penalty to update redundant info

# Redundant Arrays of Inexpensive Disks

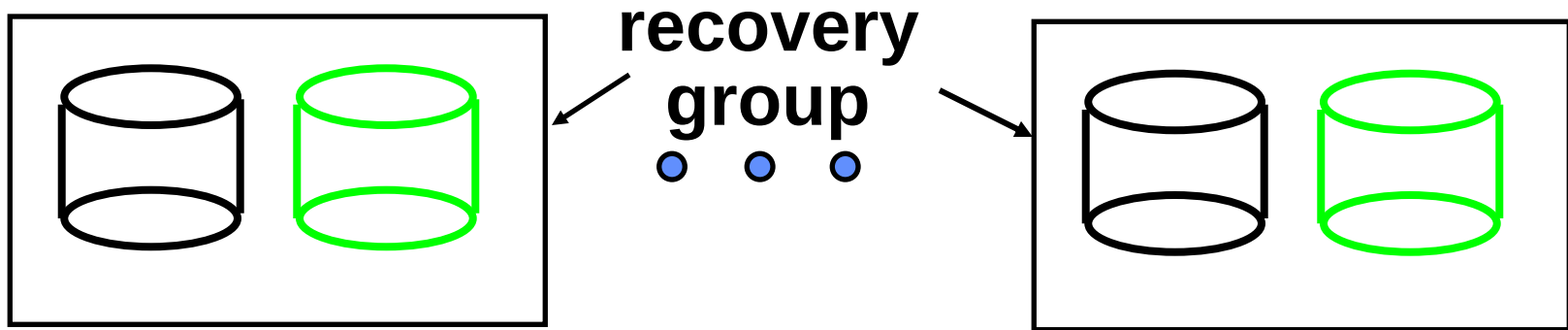
## RAID 0: Striping



- Stripe data at the block level across multiple disks
- High read and write bandwidth
- Not a true RAID since no redundancy
- Failure of any one drive renders entire array unavailable

# Redundant Arrays of Inexpensive Disks

## RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its “mirror”
  - Can result in high availability
- Bandwidth sacrifice on write:
  - Logical write = two physical writes
    - Reads may be optimized
- Most expensive solution: 100% capacity overhead
- (RAID 2 not interesting, so skip)

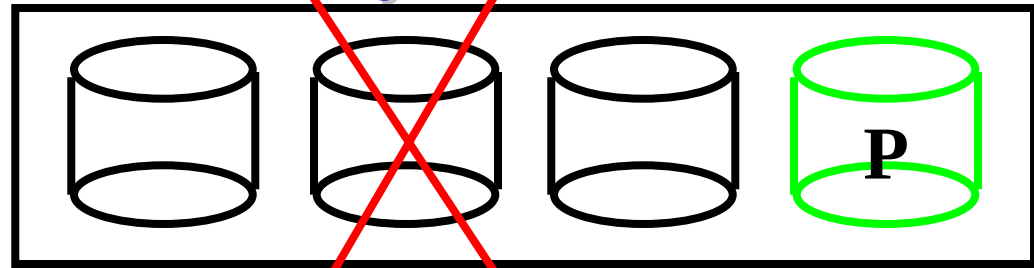
# Redundant Array of Inexpensive Disks

## RAID 3: Parity Disk

10010011
11001101
10010011
...

logical record

Striped physical  
records



1	1	1	1
0	1	0	1
1	0	1	0
0	0	0	0
0	1	0	1
0	1	0	1
1	0	1	0
1	1	1	1

P contains sum of  
other disks per stripe  
mod 2 (“parity”)

If disk fails, subtract  
P from sum of other  
disks to find missing information

## RAID 3

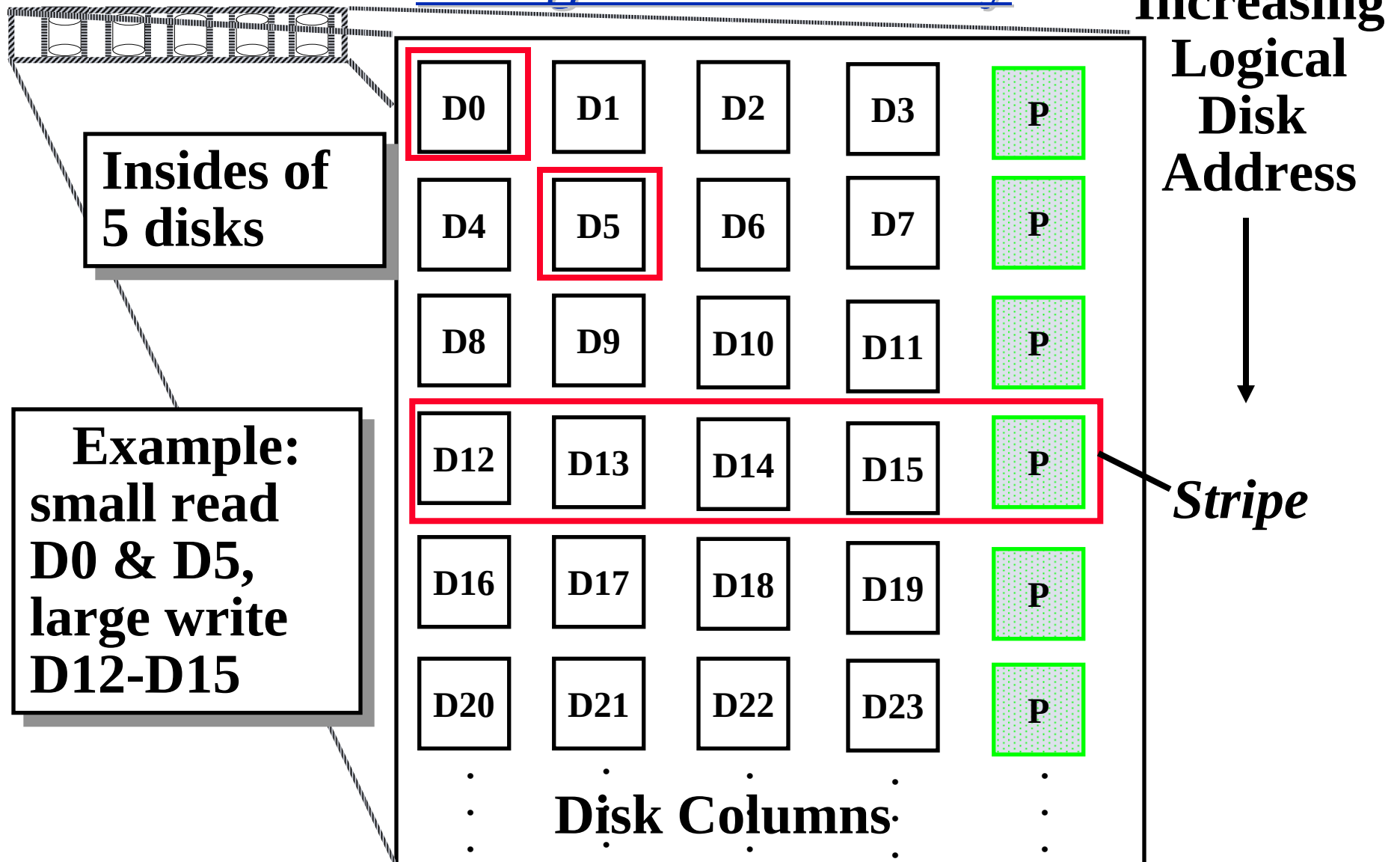
- Sum computed across recovery group to protect against hard disk failures, stored in P disk
- Logically, a single high capacity, high transfer rate disk: good for large transfers
- Wider arrays reduce capacity costs, but decreases availability
- 33% capacity cost for parity in this configuration

## Inspiration for RAID 4

- RAID 3 relies on parity disk to discover errors on Read
- But every sector has an error detection field
- Rely on error detection field to catch errors on read, not on the parity disk
- Allows independent reads to different disks simultaneously

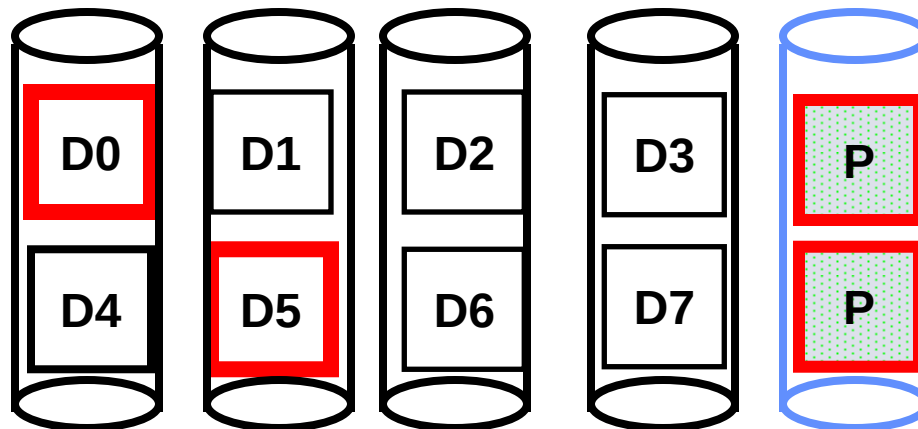
# Redundant Arrays of Inexpensive Disks RAID

## 4: High I/O Rate Parity



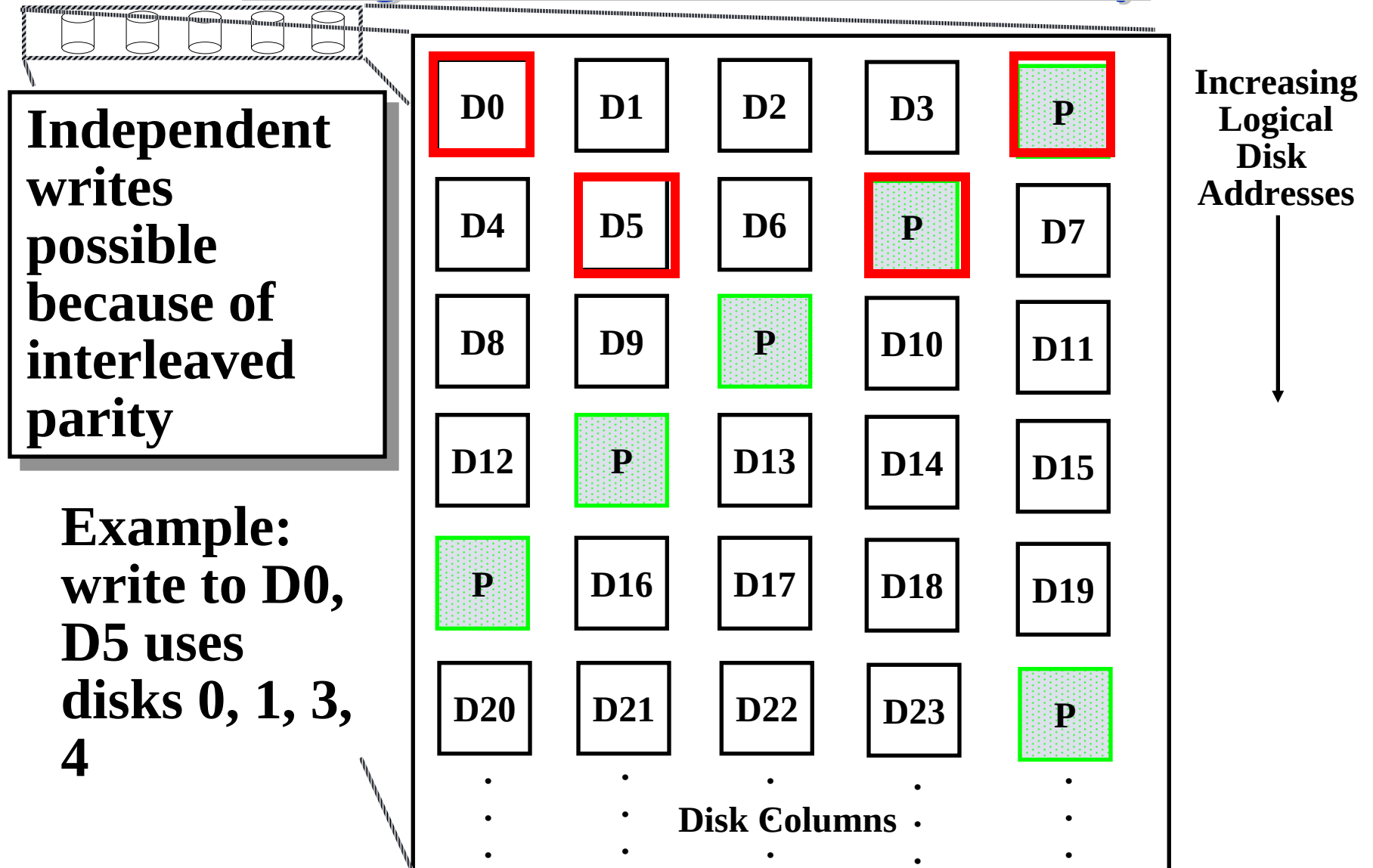
## Inspiration for RAID 5

- RAID 4 works well for small reads
- Small writes (write to one disk):
  - Option 1: read other data disks, create new sum and write to Parity Disk
  - Option 2: since P has old sum, compare old data to new data, add the difference to P
- Small writes are limited by Parity Disk: Write to D0, D5 both also write to P disk



# Redundant Arrays of Inexpensive Disks RAID

## 5: High I/O Rate Interleaved Parity

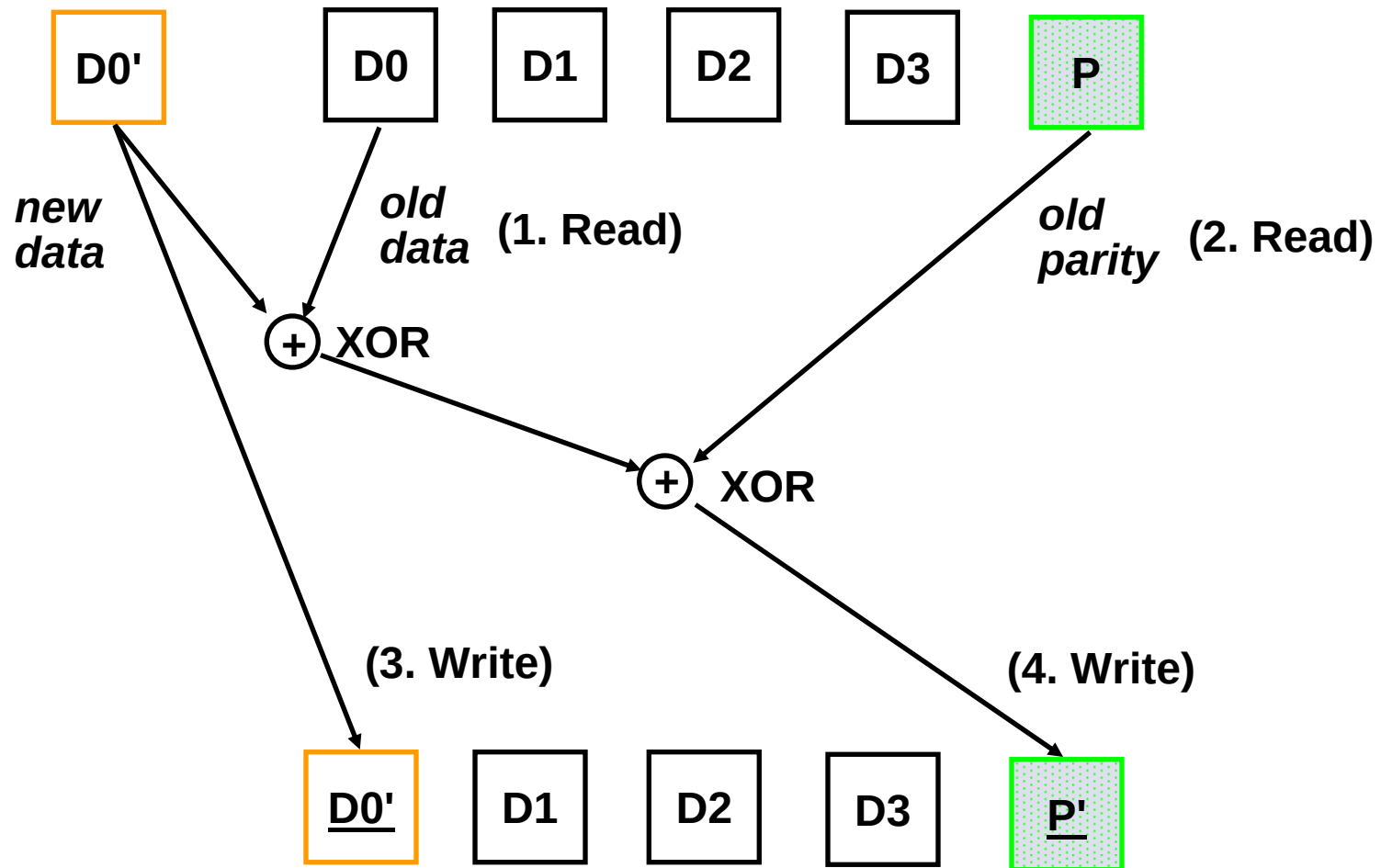


# Problems of Disk Arrays:

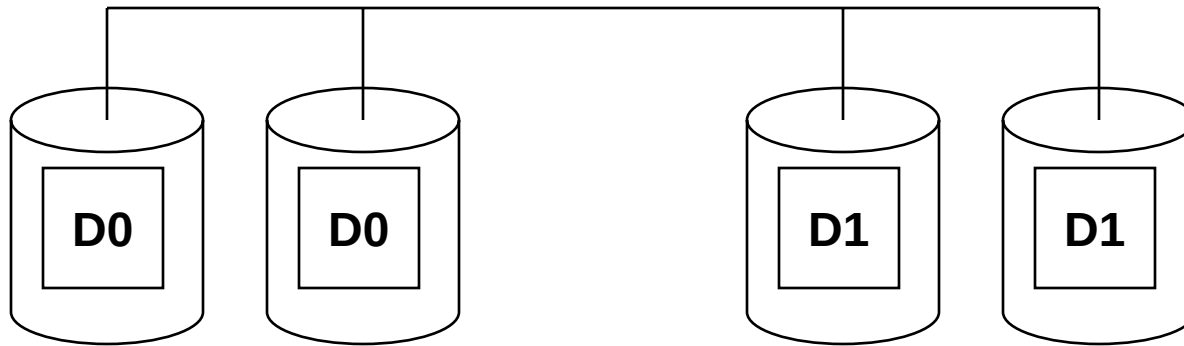
## Small Writes

### *RAID-5: Small Write Algorithm*

1 Logical Write = 2 Physical Reads + 2 Physical Writes



## RAID-10



- Striping + mirroring
- High storage overhead/cost

# RAID-I

- RAID-I (1989)
  - Consisted of a Sun 4/280 workstation with 128 MB of DRAM, four dual-string SCSI controllers, 28 5.25-inch SCSI disks and specialized disk striping software

