Distributed Systems
CS 111
Summer 2025
Operating System Principles
Peter Reiher

Outline

- Introduction
- Distributed system paradigms

Introduction

- Why do we care about distributed systems?
  - Because that's how most modern computing is done
- Why is this an OS topic?
  - Because it's definitely a systems issue
  - And even the OS on a single computer needs to worry about distributed issues
- If you don't know a bit about distributed systems, you're not a modern computer scientist

Why Distributed Systems?

- Better scalability and performance
  - Apps require more resources than one computer has
  - Can we grow system capacity/bandwidth to meet demand?
- Improved reliability and availability
  - 24x7 service despite disk/computer/software failures
- Ease of use, with reduced operating expenses
  - Centralized management of all services and systems
  - Buy (better) services rather than computer equipment
- Enabling new collaboration and business models
  - Collaborations that span system (or national) boundaries
  - A global free market for a wide range of new services

A Few Little Problems

- Different machines don't share memory
  – Or any peripheral devices
  – So one machine can't easily know the state of another

Might this cause synchronization problems?

- The only way to interact remotely is to use a network

So how can we know what's going on remotely?

  – Usually asynchronous, slow, and error prone
  – Usually not controlled by any single machine
- Failures of one machine aren't visible to other machines

How can our computation be reliable if pieces fail?

Transparency

- Ideally, a distributed system would be just like a single machine system
- But better
  - More resources
  - More reliable
  - Faster
- *Transparent* distributed systems look as much like single machine systems as possible

Deutsch's "Seven Fallacies of Network Computing"

1. The network is reliable
2. There is no latency (instant response time)
3. The available bandwidth is infinite
4. The network is secure
5. The topology of the network does not change
6. There is one administrator for the whole network
7. The cost of transporting additional data is zero

Bottom Line: true transparency is not achievable

Here's an eight: all locations on the network are equivalent.

Distributed System Paradigms

- Parallel processing
  - Relying on tightly coupled special hardware
- Single system images (SSIs)
  - Make all the nodes look like one big computer
  - Nice, but very difficult to make work
- Loosely coupled systems
  - Work with difficulties as best as you can
  - Typical modern approach to distributed systems
- Cloud computing
  - A recent variant

Parallel Processing Systems

- Systems designed to work on special hardware platforms
- Such hardware consists of several semi-independent computers
  - Usually called *nodes*
- Connected by dedicated communications medium
  - Either hard-wired links
  - Or a memory bus connected to some shared RAM
- Intended to run single jobs fast

Using Parallel Processors

- Typically a job is designed for running on this type of parallel processor
- The jobs is divided into sub-components, each run on one node of the machine
- The sub-components communicate with each other when needed, over the shared medium
- Some form of synchronization of communication is provided

Parallel Processor Hardware Example

- Hypercubes
  - Some number of independent nodes are organized into n-dimensional cubes
  - E.g., 16 nodes in a 4x4 cube
  - Links between adjacent nodes carry messages
- BBN Butterfly
  - Independent nodes share a common memory
  - Via a special shared bus
  - Which synchronizes nodes' access to the memory

Problems With Parallel Processors

- They require specially designed hardware
  - Which always turns out to be behind the design curve
  - So the nodes are slower than the best single processor you can buy
  - Very expensive
- Hard to design applications that actually run a lot faster
  - The hardware could be faster in principle
  - But synchronization issues slow things down
  - Either in shared communication medium
  - Or in application synchronization

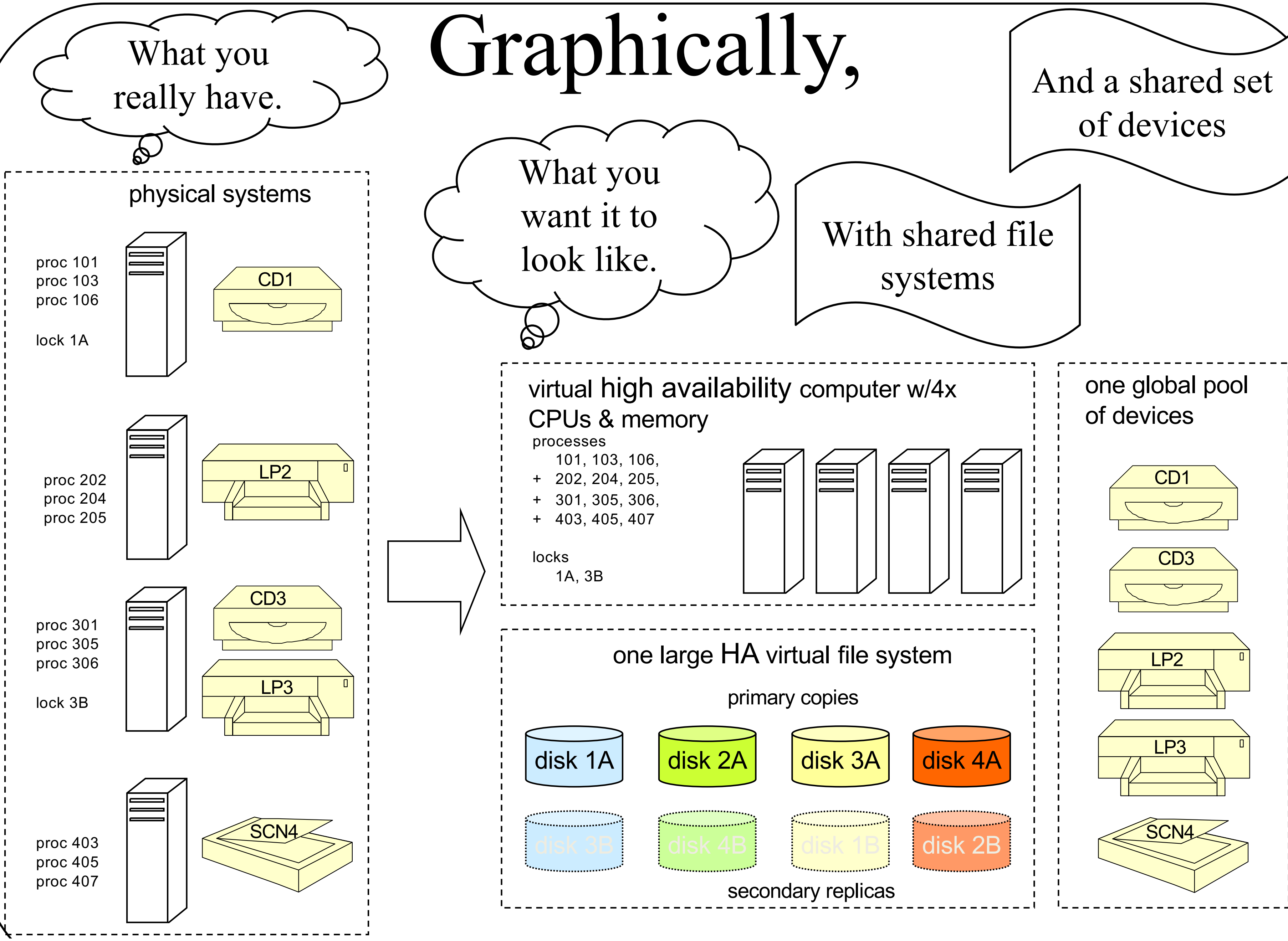
Single System Image Approaches

- A group of seemingly independent computers collaborating to provide high transparency
- Motivation:
  - Higher reliability, availability than single machines
  - More scalable than parallel processors
  - Excellent application transparency
- Examples:
  - Locus, Sun Clusters, MicroSoft Wolf-Pack, OpenSSI

The Goal

- Programs don’t run on hardware, they run atop operating systems
- All the resources that processes see are already virtualized
- Instead of merely virtualizing all the resources in a single system, virtualize all the resources in a cluster of systems.
- Applications that run in such a cluster are (automatically and transparently) distributed

Graphically,



OS Design for SSI Clustering

- All nodes agree on the state of all OS resources
  - File systems, processes, devices, locks IPC ports
  - Any process can operate on any object, transparently
- They achieve this by exchanging messages
  - Advising one-another of all changes to resources
    - Each OS's internal state mirrors the global state
  - Requesting execution of node-specific requests
    - Node-specific requests are forwarded to owning node

Networking for SSI Systems

- There will be a lot of messages
  – Accessing remote files
  – Coordinating system activities
  – Using remote devices
- So you need fast, reliable networking
- Most suitable for local area networks
- Running this over the Internet is challenging

Reliability Issues

- One goal is to provide continued service even when single machines fail
  - Obviously limited by stuff that’s on the failed machine
  - E.g., files hosted on its storage devices
- Definite advantages when failures are simple
- Things get complex if the network partitions
  - Divides into pieces that can’t talk to each other
- Or when things keep coming and going

SSI Clustered Performance

- Clever implementation can minimize overhead
  - 10-20% overall is common, can be much worse
- Complete transparency
  - Even very complex applications “just work” (mostly . . .)
- Good robustness
  - When one node fails, others notice and take-over
  - Often, applications won't even notice the failure
- Nice for application developers and customers
- But implementation is large, complex, difficult, and not scalable
- The exchange of messages can be very expensive

Lessons Learned From SSI Research

- Consensus protocols are expensive
  - They converge slowly and scale poorly
- Systems have a great many resources
  - Resource change notifications are expensive
- Location transparency encouraged non-locality
  - Remote resource use is much more expensive
- A greatly complicated operating system
  - Distributed objects are more complex to manage
  - Complex optimizations to reduce the added overheads
  - New modes of failure with complex recovery procedures

Loosely Coupled Systems

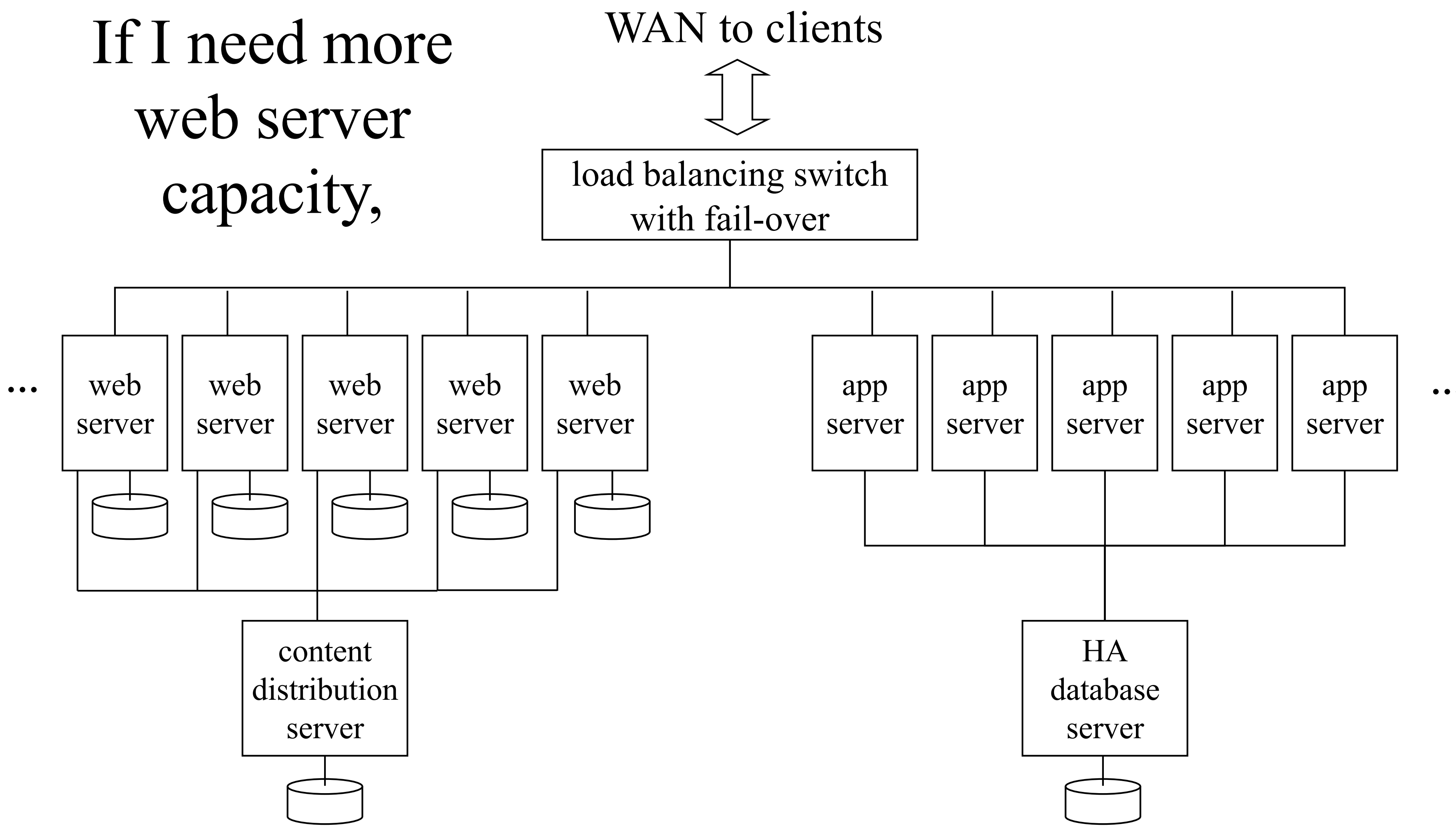
- Characterization:
  - A parallel group of independent computers
  - Connected by a high-speed LAN
  - Serving similar but independent requests
  - Minimal coordination and cooperation required
- Motivation:
  - Scalability and price performance
  - Availability – if protocol permits stateless servers
  - Ease of management, reconfigurable capacity
- Examples:
  - Web servers, app servers

Horizontal Scalability

- Each node largely independent
- So you can add capacity just by adding a node "on the side"
- Scalability can be limited by network, instead of hardware or algorithms
  - Or, perhaps, by a load balancer
- Reliability is high
  - Failure of one of N nodes just reduces capacity

Horizontal Scalability Architecture

If I need more web server capacity,



Elements of Loosely Coupled Architecture

- Farm of independent servers
  - Servers run same software, serve different requests
  - May share a common back-end database
- Front-end switch
  - Distributes incoming requests among available servers
  - Can do both load balancing and fail-over
- Service protocol
  - Stateless servers and *idempotent* operations
  - Successive requests may be sent to different servers

Same result if you do it once, twice, three times, . . ., *n* times

Horizontally Scaled Performance

- Individual servers are very inexpensive
  - Blade servers may be only $100-$200 each
- Scalability is excellent
  - 100 servers deliver approximately 100x performance
- Service availability is excellent
  - Front-end automatically bypasses failed servers
  - Stateless servers and client retries fail-over easily
- The challenge is managing thousands of servers
  - Automated installation, global configuration services
  - Self monitoring, self-healing systems
  - Scaling limited by management, not HW or algorithms

Advantages and Disadvantages

+ Highly practical
+ Much simpler problems to handle than with SSI or parallel processing
+ Allows use of cheap hardware
+ Very scalable in many dimensions
+ A good match for many important applications
- Not such a good match for many others
- Some scaling limitations based on shared elements (e.g., load balancers)

Cloud Computing

- The most recent twist on distributed computing
- We discussed this a couple lectures back
- What runs in a cloud?
- In principle, anything
  – But general distributed computing is hard
- So much of the work is run using special tools
- These tools support particular kinds of parallel/distributed processing
  – Using a method like map-reduce or horizontal scaling
- So the user need not be a distributed systems expert

MapReduce

- Perhaps the most common cloud computing software tool/technique
- A method of dividing large problems into compartmentalized pieces
- Each of which can be performed on a separate node
- With an eventual combined set of results

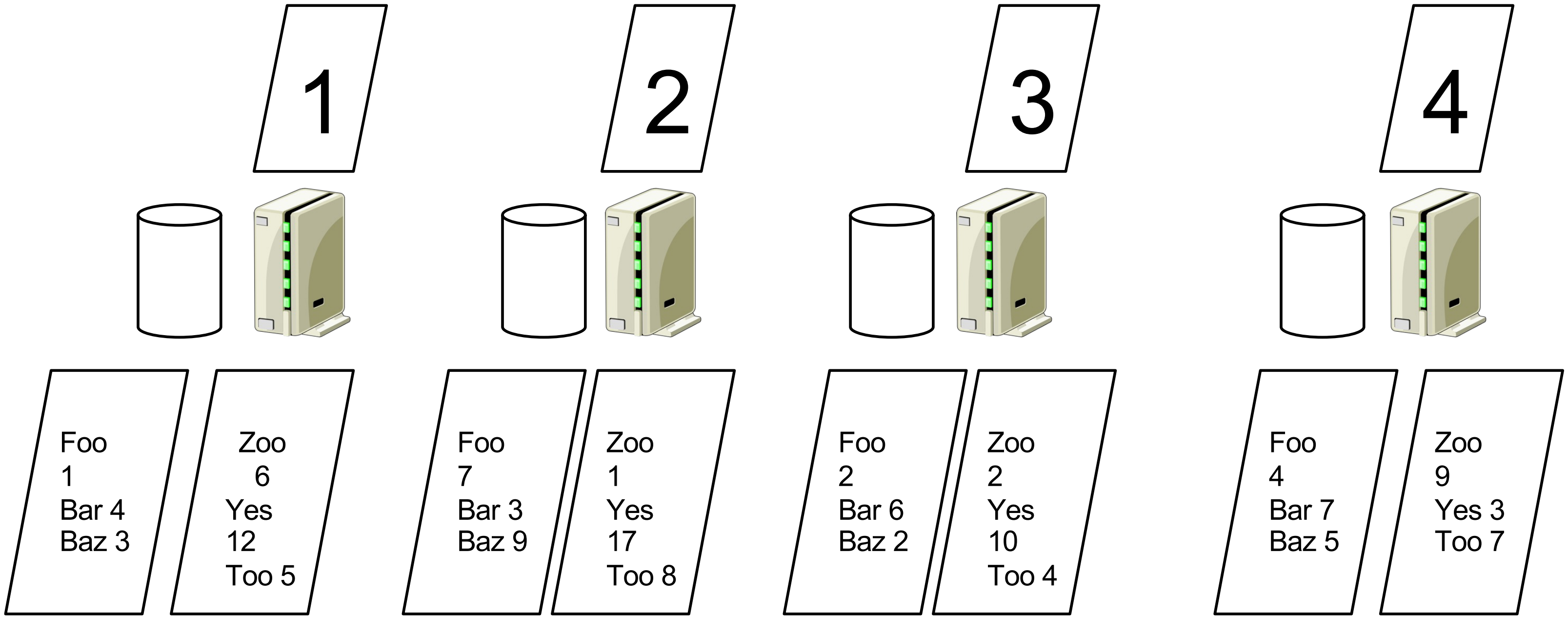
The Idea Behind MapReduce

- There is a single function you want to perform on a lot of data
  - Such as searching it for a particular string
- Divide the data into disjoint pieces
- Perform the function on each piece on a separate node (*map*)
- Combine the results to obtain output (*reduce*)

An Example

- We have 64 megabytes of text data
- Count how many times each word occurs in the text
- Divide it into 4 chunks of 16 Mbytes
- Assign each chunk to one processor
- Perform the map function of "count words" on each

The Example Continued

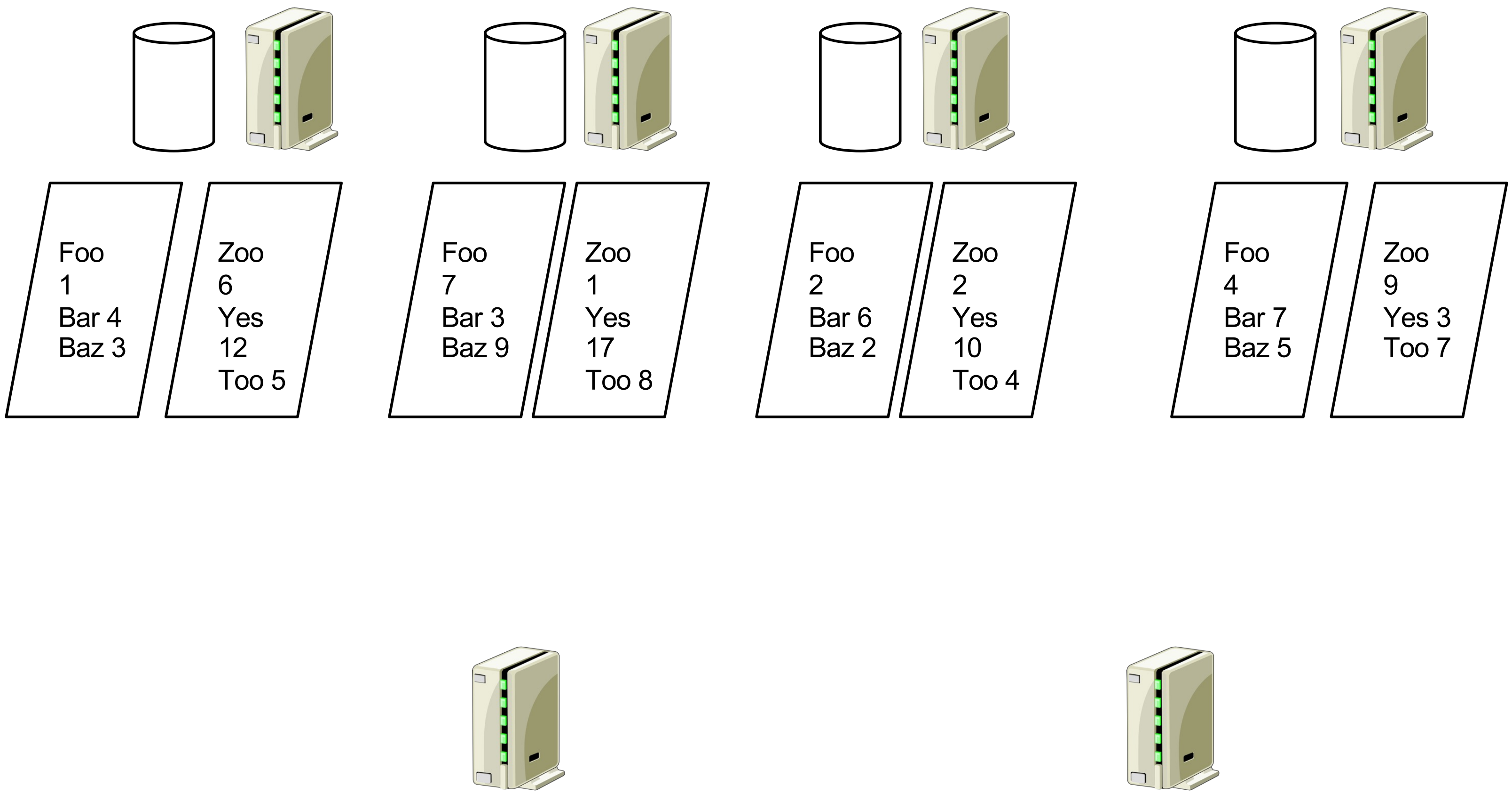


That's the map stage

On To Reduce

- We might have two more nodes assigned to doing the reduce operation
- They will each receive a share of data from a map node
- The reduce node performs a reduce operation to “combine” the shares
- Outputting its own result

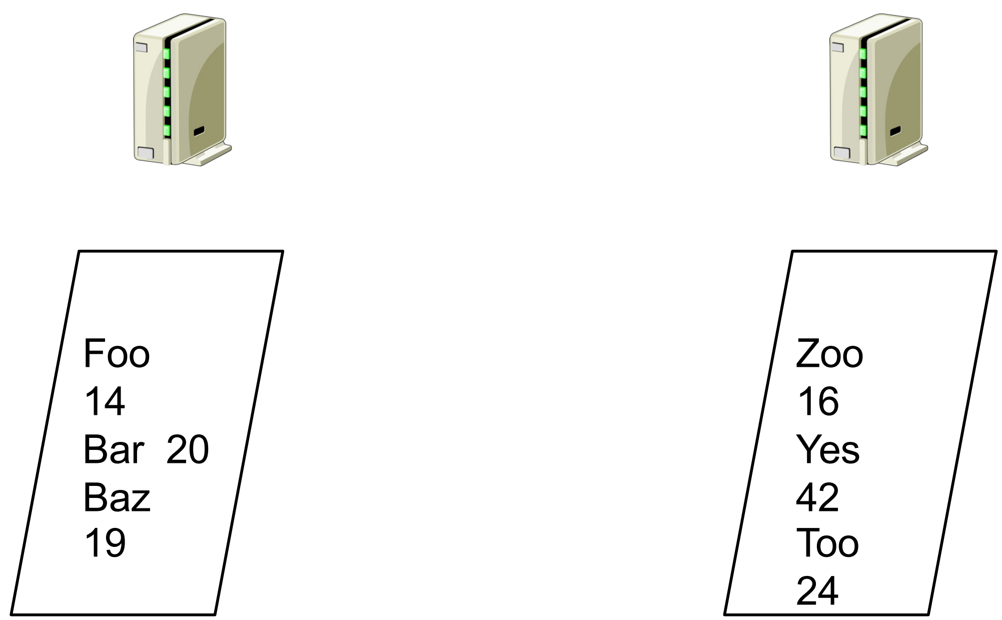
Continuing the Example



The Reduce Nodes Do Their Job

Write out the results to files

And MapReduce is done!



But I Wanted A Combined List

- No problem
- Run another (slightly different) MapReduce on the outputs
- Have one reduce node that combines everything

Synchronization in MapReduce

- Each map node produces an output file for each reduce node
- It is produced atomically
- The reduce node can't work on this data until the whole file is written
- Forcing a synchronization point between the map and reduce phases
- Also resilient to many partial failures
  - Easy detection and recovery are possible

Cloud Computing and Horizontal Scaling

- An excellent match
- Rent some cloud nodes to be your web servers
- If load gets heavy, ask the cloud for another web server node
- As load lightens, release unneeded nodes
- No need to buy new machines
- No need to administer your own machines

Cloud Computing Advantages and Disasadvantages

+ Hides much complexity from end users
+ Good match for many important computing uses
+ Allows excellent scalability for users
+ Strong economic model for both users and providers
- Doesn’t help with some problems
- Expensive to create and operate

Remote Procedure Calls

- RPC, for short
- One way of building a distributed program
- Procedure calls are a fundamental paradigm
  - Primary unit of computation in most languages
  - Unit of information hiding in most methodologies
  - Primary level of interface specification
- RPC allows procedures on one machine to call procedures on a different machine
  - Looking largely like ordinary procedure calls

RPC Characteristics

- Procedure calls are a natural boundary between client and server
  - Context in the called procedure largely hidden from caller
- RPC turns procedure calls into message send/receives
  - Calling procedure sends a message to machine hosting remote procedure
  - That machine returns the result in a message

RPC Limitations

- But local procedure calls don't perfectly match message sends and receives
- So RPC has a few limitations
  - No implicit parameters/returns (e.g., global variables)
  - No call-by-reference parameters (i.e., passing pointers in parameters)
  - Much slower than procedure calls (TANSTAAFL)
- Another limitation - RPC doesn't make most distributed systems problems disappear

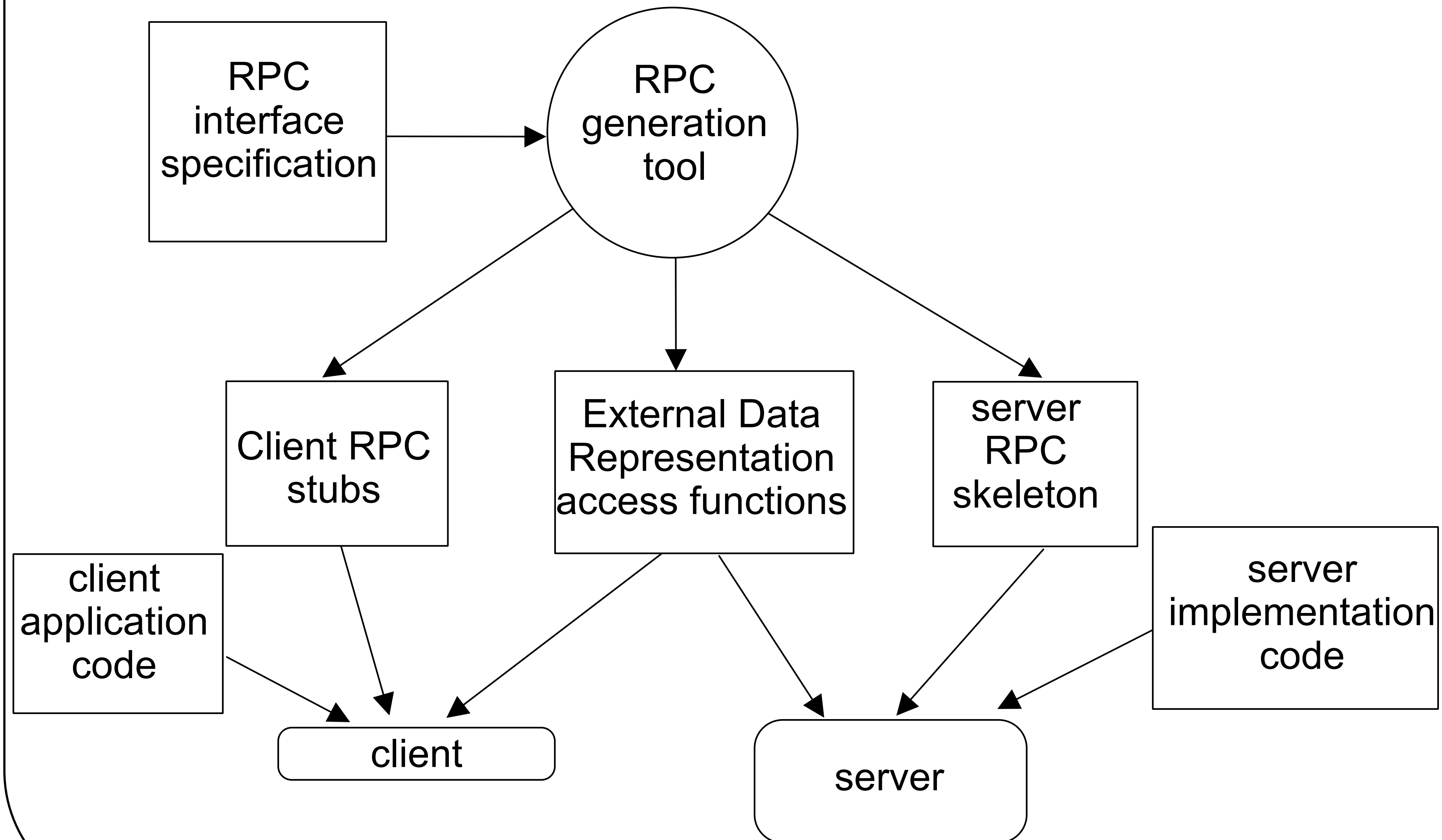
Using RPC

- Typically supported by a library or package
  - Which hides messy details of converting procedure calls to messages
  - Callers and procedure providers must use the same library
- Also by processes on the site hosting the remote procedures
  - To receive messages and invoke the procedures

Remote Procedure Call Concepts

- Interface Specification
  - Methods, parameter types, return types
- eXternal Data Representation (XDR)
  - Machine independent data-type representations
  - May have optimizations for similar client/server
- Client stub
  - Client-side proxy for a method in the API
- Server stub (or skeleton)
  - Server-side recipient for API invocations

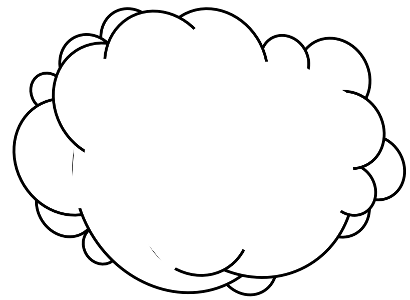
RPC Tool Chain



Key Features of RPC

- Client application links against local procedures
  - Calls local procedures, gets results
- All RPC implementation inside those procedures
- Client application does not know about RPC
  - Does not know about formats of messages
  - Does not worry about sends, timeouts, resends
  - Does not know about external data representation
- All of this is generated automatically by RPC tools
- The key to the tools is the interface specification

RPC At Work, Step 1

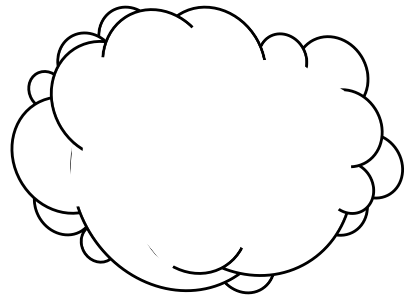


`Process_list <return>`

```
   . . .
list[0] = 10;
list[1] = 20;
list[2] = 17;
max = list_max(list);
```

`list_max()` is a remote procedure call!

RPC At Work, Step 2



```
Process_list <return>
```

```
   . . .
list[0] = 10;
list[1] = 20;
list[2] = 17;
max = list_max(list);
```

Format RPC message

RPC message: `list_max()`, parameter `list`

Send the message

```
local_max =
list_max(list);
```

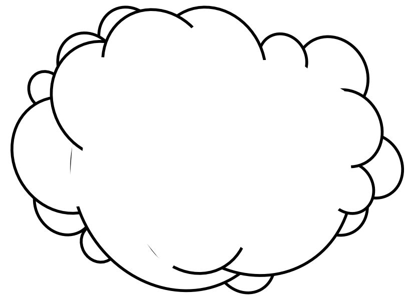
Extract RPC info

`list_max()`

`list`

Call local procedure

RPC At Work, Step 3



```
   . . .
list[0] = 10;
list[1] = 20;
list[2] = 17;
max = list_max(list);
If (max > 10) {
```

max 20

Extract the return value

Resume the local program

```
local_max =
list_max(list);
```

local_max 20

Format RPC response

RPC response: `list_max()`, return value `20`

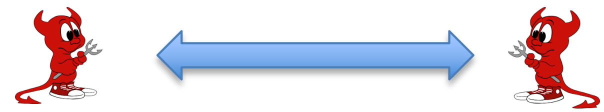
Send the message

Practical Use of RPC

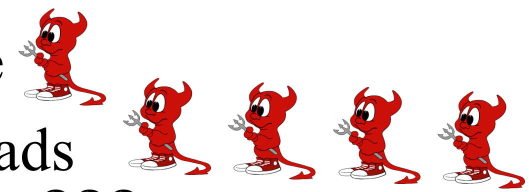
- In concept, servers providing RPC could support arbitrary clients
  - Without knowledge of what the clients are doing
- In practice, RPC is usually used to build a specific distributed system
  - Designed to support one (typically large) application
  - Which by nature must run on many machines
  - Like a specialized system for a large company

RPC Is Not a Complete Solution

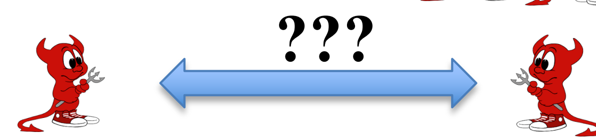
- Requires client/server binding model



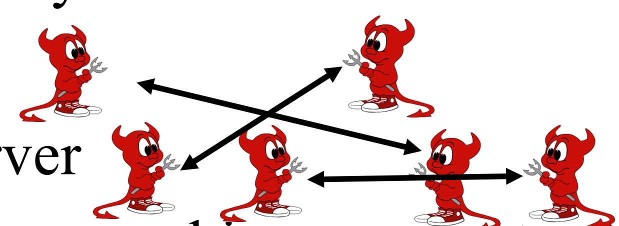
  - Expects to be given a live connection
- Threading model implementation for RPC servers
  - A single thread services requests one at a time



  - So use numerous one-per-request worker threads
- Limited failure handling



  - Client must arrange for timeout and recovery
- Limited consistency support



  - Only between calling client and called server
  - What if there are multiple clients and servers working together?
- Higher level abstractions improve RPC
  - e.g. Microsoft DCOM, Java RMI, DRb, Pyro

Conclusion

- Distributed systems offer us much greater power than one machine can provide
- They do so at costs of complexity
- We handle the complexity by using distributed systems in a few carefully defined ways