

Chapter 2

Principles of Parallel and Distributed Computing

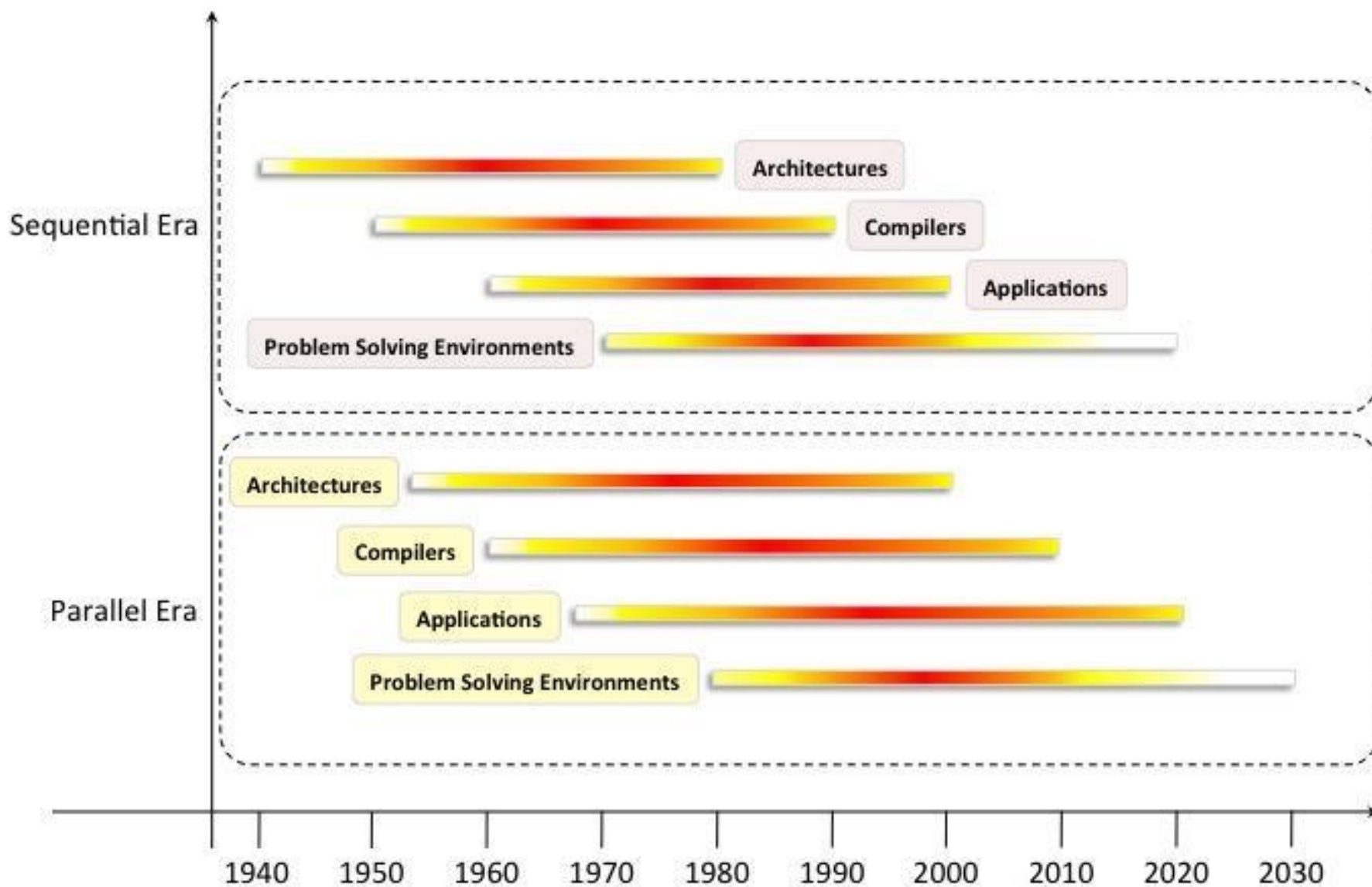
Mastering Cloud Computing
Paul Talaga

Computing Eras

- Sequential - 1940s+
- Parallel and Distributed - 1960s+

But... CS typically teaches sequential
Parallel programming is hard!

Moore's Law (modern CPUs) now require
us into program parallel programs.



Parallel vs. Distributed

- Parallel - tightly coupled system
 - Computation divided among processors sharing common memory
 - Homogenous components
 - Defn loosening with [InfiniBand](#) & dist mem
- Distributed - any system where computation is broken down and executed concurrently
 - parallel a subtype - distributed more general
 - Different nodes, processors, or cores
 - heterogeneous components

Parallel Computing

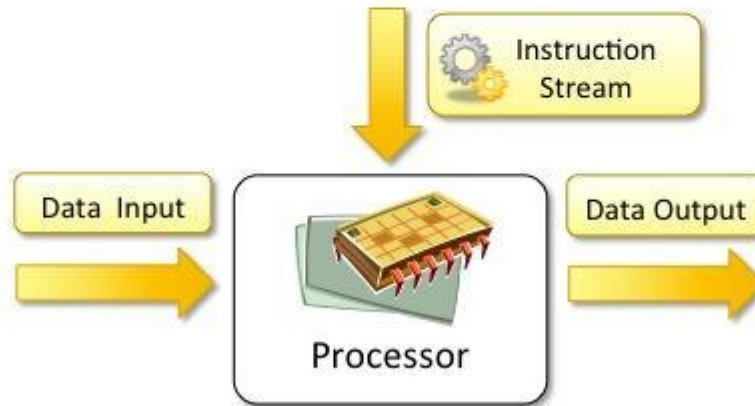
Renewed interest...

- Larger computation tasks
- CPU have reached physical limits
- Hardware features (pipelining, superscalar, etc) require complex compilers - reached limits
- Vector processing effective, but applicability isolated
- Networking technology mature

Hardware Architectures

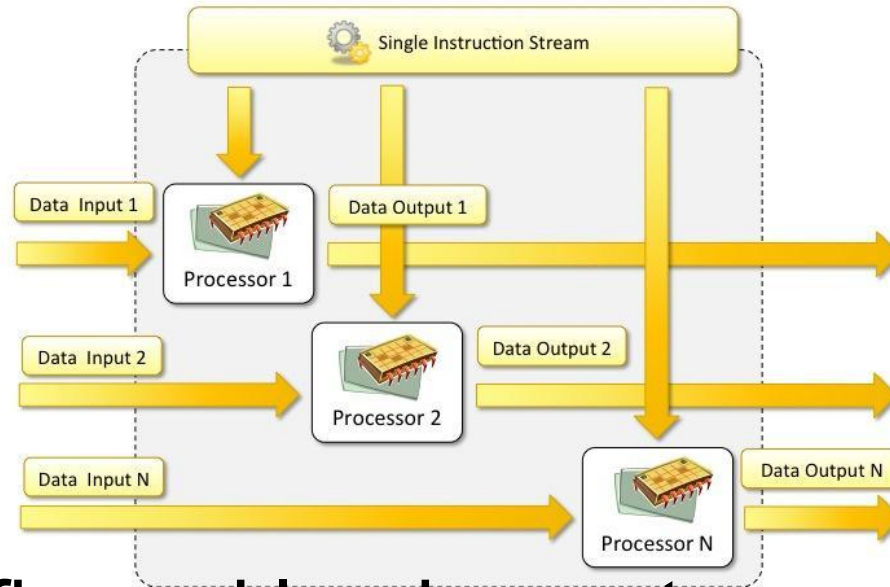
- Single instruction, single data (SISD)
- Single instruction, multiple data (SIMD)
- Multiple instruction, single data (MISD)
- Multiple instruction, multiple data (MIMD)

Single instruction, single data (SISD)



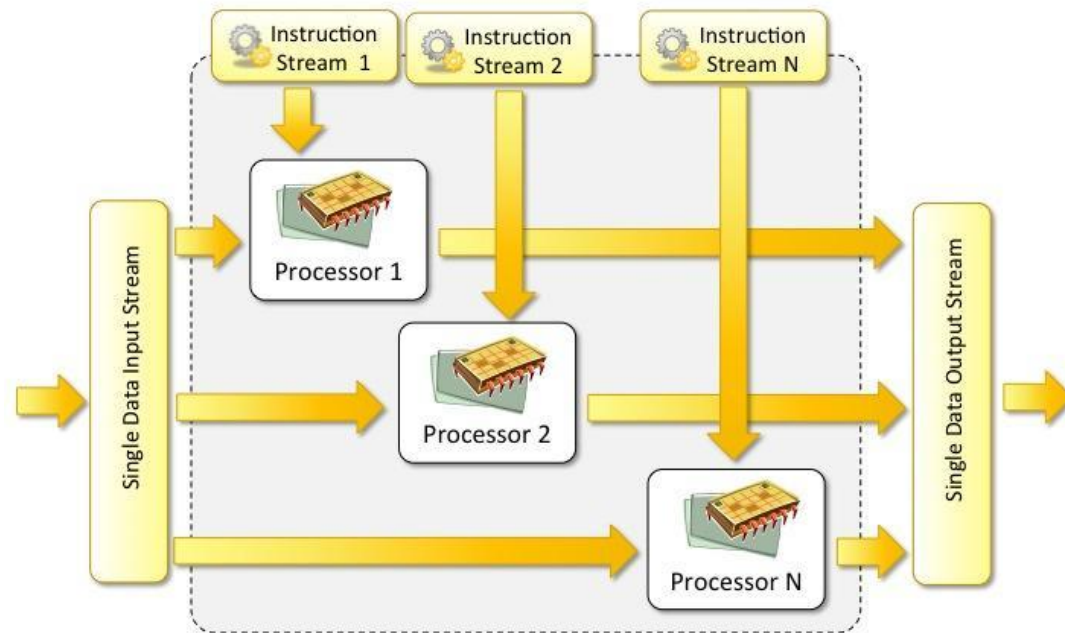
- Sequential computers
- 'Normal' computers, PCs, Macs
- CS1, CS2, DS - typically programming

Single instruction, multiple data (SIMD)



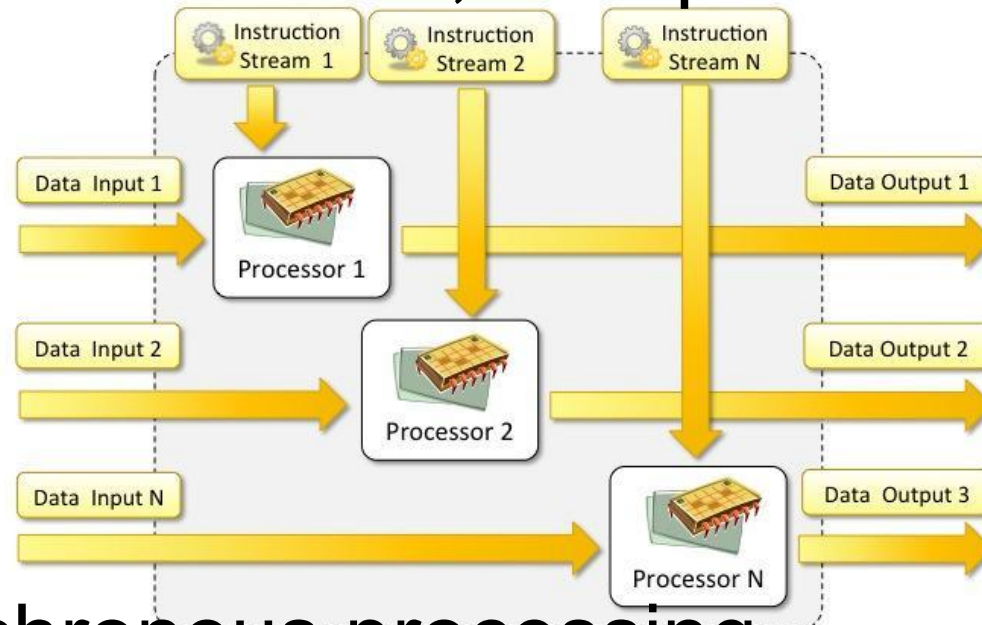
- Scientific workloads, vector and matrix operations
- GPUs ([CUDA](#)), Sony PS3 Cell processor ([1,2](#)), Cray's vector processor, Thinking Machines' cm*

- Multiple instruction, single data (MISD)



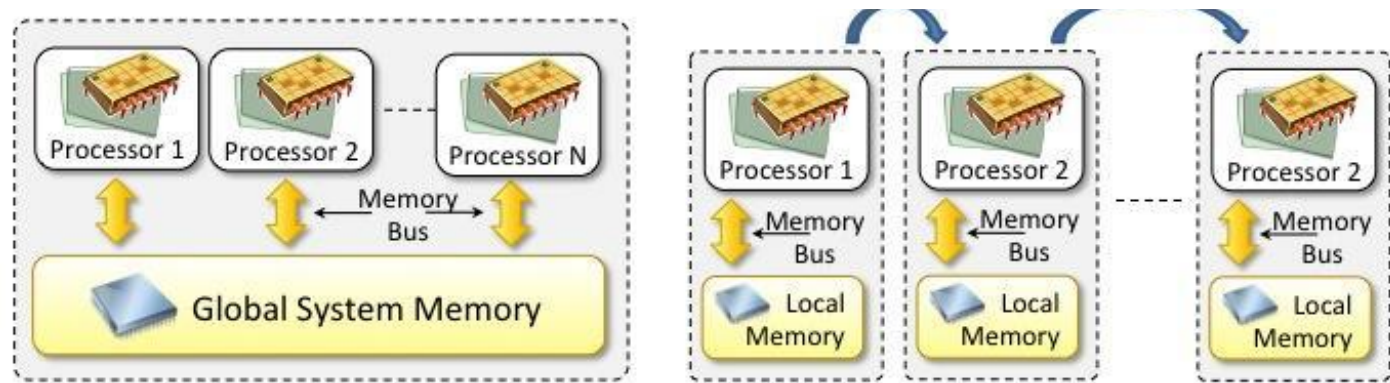
- $y = \sin(x) + \cos(x) + \tan(x)$
- No commercial machines exist, though CPU superscalar and pipelining have a similar feel

Multiple instruction, multiple data (MIMD)



- Asynchronous processing
- 2 types:
 - shared-memory
 - Silicon Graphics (80s, 90s) ([1](#))
 - Sun/IBM's SMP
 - distributed-memory

Memory Architectures in MIMD



- Shared - L1/L2/L3/Main memory in multicore - cache consistency hard/slow - not scalable
- Distributed - fewer consistency issues, but messages needed - popular

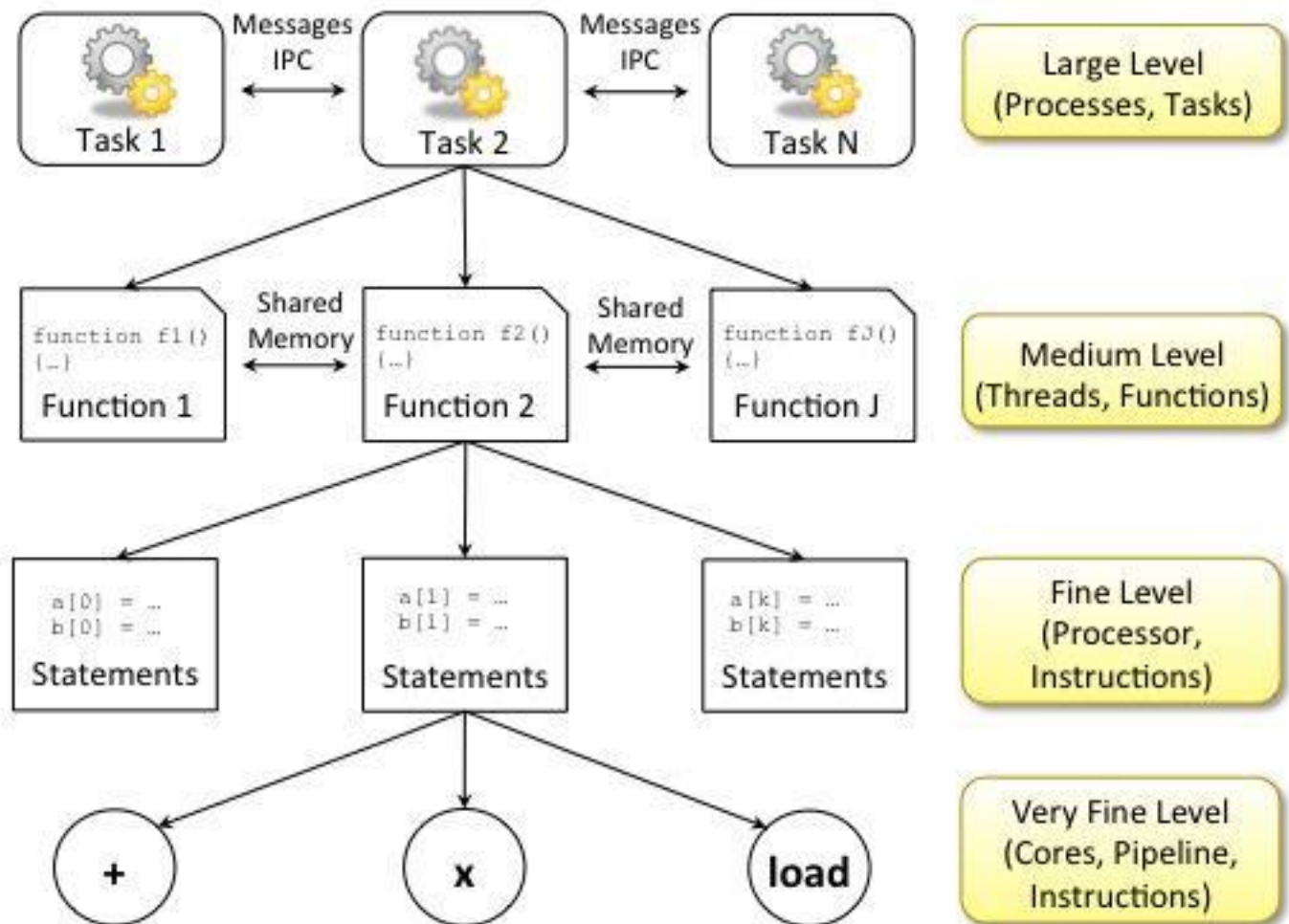
How to Program in Parallel?

- Problem specific! (Ug)
- Approaches:
 - Data parallelism
 - MapReduce
 - Process parallelism
 - Game/Cell Processor
 - Farmer-and-worker model
 - Web serving (Apache)

Another Consideration: Level of Parallelism

Goal? Never have a processor idle!
'Grain size' important - how you break up
the problem.

Grain Size	Code Item	Parallelized By
Large	Separate (heavyweight process)	Programmer
Medium	Function or procedure (thread)	Programmer
Fine	Loop or instruction block	Compiler
Very fine	Instruction	Processor or OS



Linear speedup not possible

- Doubling # cores doesn't double speed
 - Communication overhead
- General guidelines:
 - Computation speed = $\sqrt{\text{system cost}}$ or faster a system becomes the more expensive it is to make it faster
 - Speed of parallel computer increases as the log of # of processors

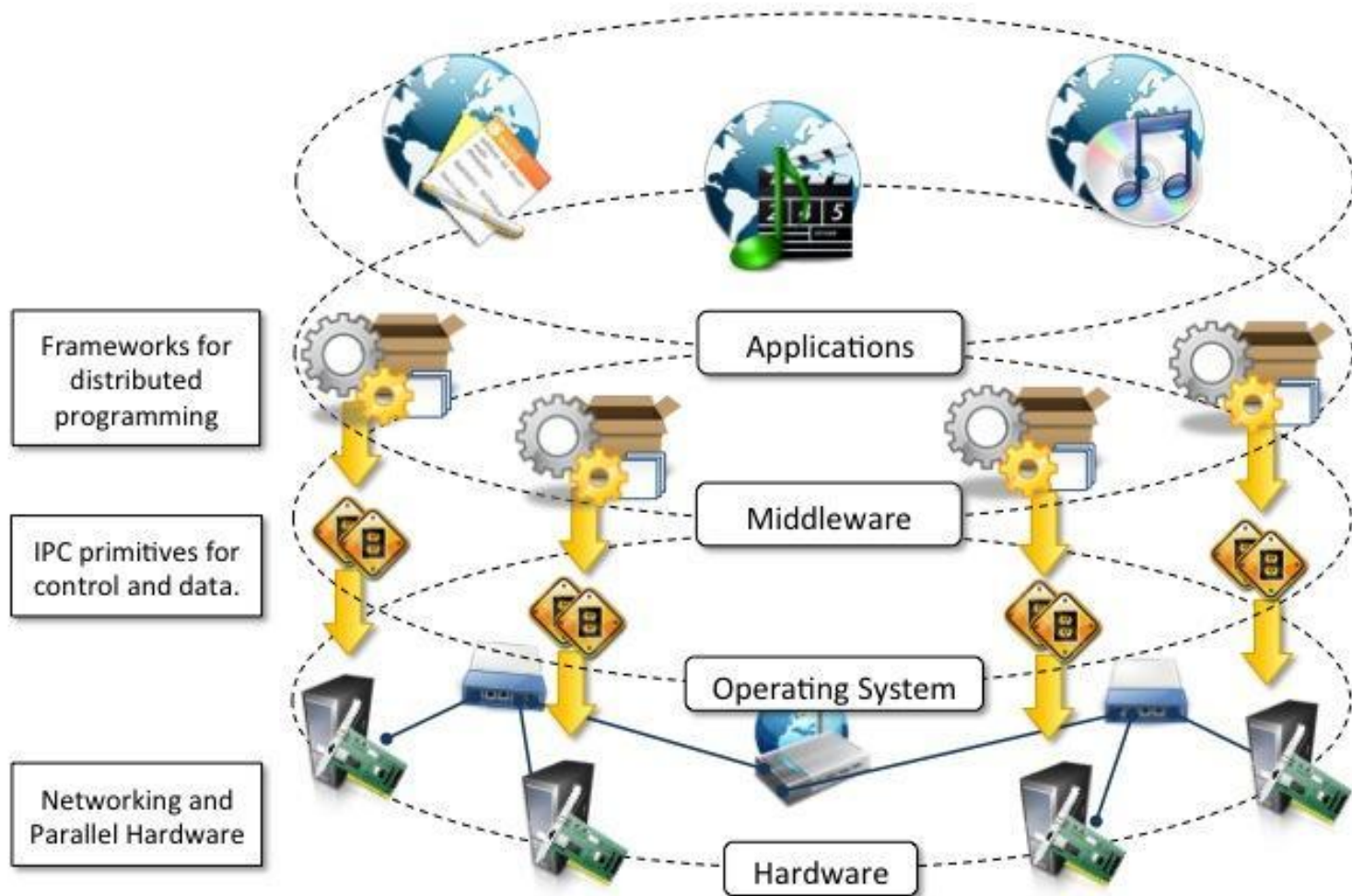
Elements of Distributed Computing

A distributed system is a collection of independent computer that appears to its users as a single coherent system - Tanenbaum et al

A distributed system is one in which components located at networked computers communicate and coordinate their actions by passing messages. - Coulouris et al

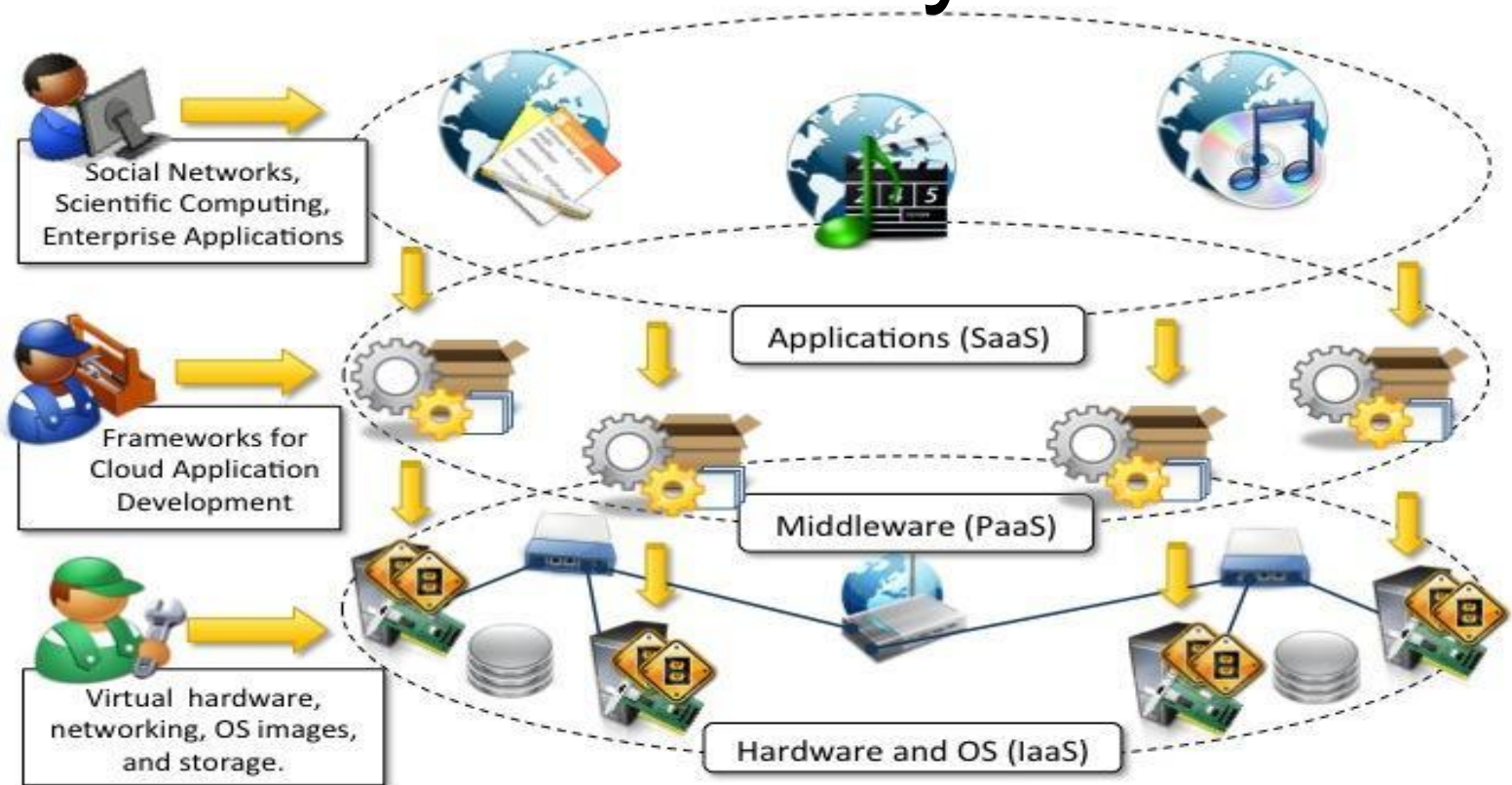
Message passing!!!!

Distributed System Layers



Abstraction!

Cloud Computing as a distributed system



Middleware enables distributed computing

Architectural Styles for Distributed Computing

Architectural styles are mainly used to determine the vocabulary of components and connectors that are used as instances of the style together with a set of constraints on how they can be combined.

Like design patterns for a program, but for an entire software system.

- Software architectural styles (software organization)
- System architectural styles (physical organization)

Components and Connectors

- Component - software that encapsulates a function or feature of the system: programs, objects, processes.
- Connector - communication mechanism between components, can be implemented in a distributed manner.


Software Architectural Styles

Category	Common Architectural Styles
Data-centered	Repository
	Blackboard
Data flow	Pipe and filter
	Batch sequential
Virtual Machine	Rule-based system
	Interpreter
Call and return	Main program and subroutine/top down
	Object-oriented systems
Independent components	Communicating processes
	Even systems

Data centered architectures

- Data is core, and access to shared data
 - Data integrity is goal
 - Ex: Gmail, Flickr, Google search
- *Repository style* -
 - central data structure - current state
 - independent components - operate on data
 - 2 subtypes:
 - database systems - components called & act on data
 - blackboard systems - data-structure is trigger - if/then or expert-system feel - updates itself

Software Architectural Styles



Category	Common Architectural Styles
Data-centered	Repository
	Blackboard
Data flow	Pipe and filter
	Batch sequential
Virtual Machine	Rule-based system
	Interpreter
Call and return	Main program and subroutine/top down
	Object-oriented systems
Independent components	Communicating processes
	Even systems

Data-Flow architectures

- Availability of data controls, data flows through system
- 2 styles:
 - Batch Sequential - sequence of programs - must wait for previous to finish before next
 - Pipe-and-Filter - sequence of programs, but FIFO queues to start processing before previous has finished. Unix shell pipes good example

Batch Sequential vs. Pipe-and-Filter

Batch Sequential	Pipe-and-Filter
Coarse grained	Fine grained
High latency	reduced latency due to incremental processing
External access to input	localized input
No concurrency	Concurrency possible
Noninteractive	Awkward, but possible

Software Architectural Styles

Category	Common Architectural Styles
Data-centered	Repository
	Blackboard
Data flow	Pipe and filter
	Batch sequential
Virtual Machine	Rule-based system
	Interpreter
Call and return	Main program and subroutine/top down
	Object-oriented systems
Independent components	Communicating processes
	Event systems



Virtual Machine architectures

- Abstract execution environment - rule-based systems, interpreters, command-language processors
- 2 styles:
 - Rule-based - inference engine - AI - process control - network intrusion detection
 - Interpreter - interprets pseudo-program - abstracts hardware differences away - Java, C#, Perl, PHP, etc

Software Architectural Styles

Category	Common Architectural Styles
Data-centered	Repository
	Blackboard
Data flow	Pipe and filter
	Batch sequential
Virtual Machine	Rule-based system
	Interpreter
Call and return	Main program and subroutine/top down
	Object-oriented systems
Independent components	Communicating processes
	Event systems

Call & Return architectures

- Components connected via method calls
- 3 styles:
 - Top-Down: imperative programming - tree structure - hard to maintain
 - Object-Oriented: coupling between data and manipulation operations - easier to maintain - method calling requires object - consistency an issue
 - Layered Style: Abstraction layers - modular design - hard to change layers
 - Ex: OS kernels, TCP/IP stack, web applications

Software Architectural Styles

Category	Common Architectural Styles
Data-centered	Repository
	Blackboard
Data flow	Pipe and filter
	Batch sequential
Virtual Machine	Rule-based system
	Interpreter
Call and return	Main program and subroutine/top down
	Object-oriented systems
Independent components	Communicating processes
	Event systems



Independent Components Architectures

- Life cycles to components
- 2 styles:
 - Communicating Processes: good for distributed systems - concurrent - service based - IPC
 - Even Systems: components have data and manipulation, but add event registering/triggering - callbacks - like layered, but looser connections - hard to reason about correctness of interactions

System Architectural Styles

- Describe physical layout
- 2 styles:
 - Client/Server
 - Peer-to-peer

Client/Server

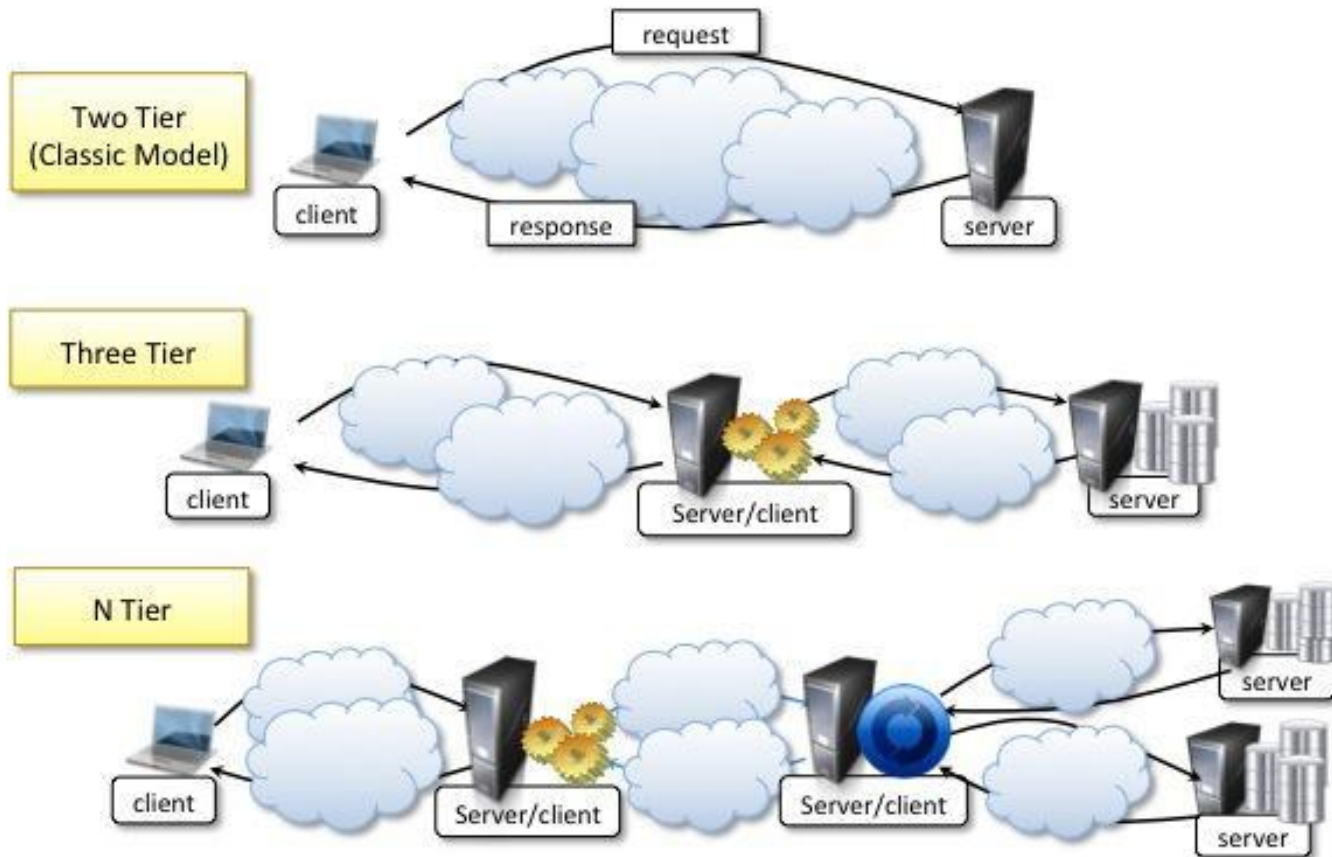
- Very Popular
- *request, accept (client)*
- *listen, response (server)*
- Suitable for many-to-one situations

Types of Clients

- Thin-client: (nearly) all data processing done on server (plain HTML)
 - presentation on client
 - app logic and data storage on server
- Fat-client: client processes and transforms data, server just gateway to access data (AJAX-like)
 - presentation and app logic on client
 - data storage on server

Layered Approach

Client-server



Layer Types

- Two Tier
 - Pros: Easier to build
 - Con: Doesn't scale well
 - Ex: Small dynamic web applications
- Three Tier/N Tier
 - Pros: Scales better (add more servers to layer)
 - Con: Harder to maintain
 - Ex: Medium-Large dynamic web applications

Peer-to-Peer



- Symmetric - everyone client and server
- Scales very well!
- Hard to build
- Used in data centers to distribute data!
- Ex: Gnutella, BitTorrent, Kazaa, Skype
- Multi-level roles possible: Kazaa

Interprocess Communication (IPC)

- How different programs/process can communicate
- Use a client/server model
- Built using network sockets
 - Allow traversal of unknown networks due to network stack

Types of Message Passing

- Message Passing: pass plain messages
 - Ex: Message-passing interface ([MPI](#)) and [OpenMP](#) - Ohio Supercomputer
- Remote Procedure Call (RPC): Run procedure/function on remote process
 - stateless
 - Ex: many many examples
 - Uses marshaling of parameters/return values

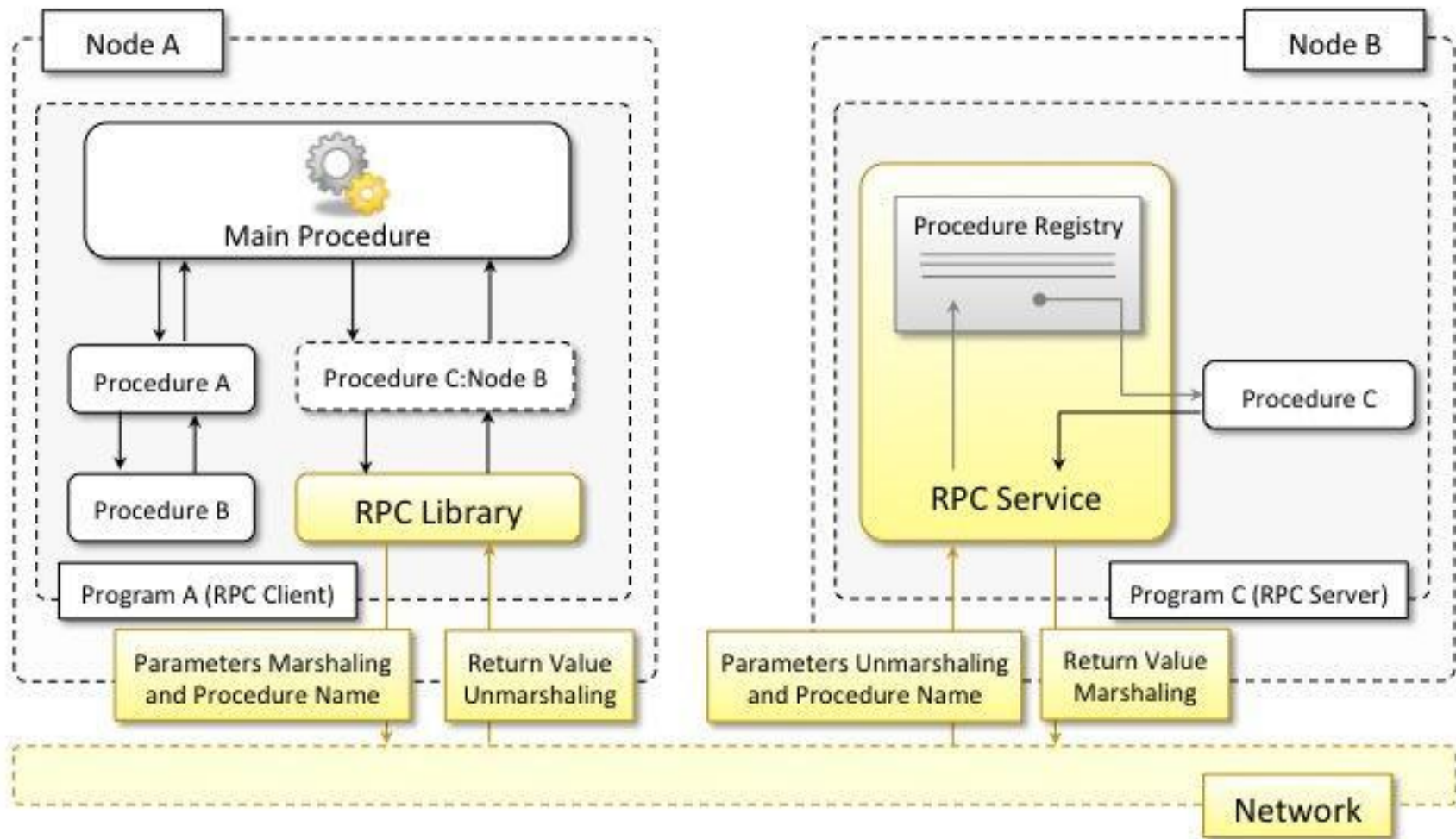
More Types of Message Passing

- Distributed objects: RPC for OO
 - remote objects have state!
 - Ex: CORBA, COM/DCOM/COM+, RMI (Java), .NET Remoting, we'll use [RPyC](#)
- Distributed agents and active objects
 - Like distributed objects, but remote instances can operate independently
- Web Services: RPC over HTTP
 - More interoperable than typical RPC
 - Ex: SOAP, REST

Models for Message Communication

- Point-to-point
 - Unique sender and receivers
 - No central infrastructure
 - request-reply model
 - 2 subcategories:
 - direct - request processed when received
 - queued - delay processing
- Publish-and-Subscribe
 - Register for notification of event
 - Push strategy - publisher notifies
 - Pull strategy - subscriber checks in for events

Technologies - RPC



RPC - Continued

- Old idea/technology (1980+)
- Caller blocks
- Marshaling/unmarshaling important:
convert parameters/return values to be
moved on network
- Pass by reference/pointers not suitable



RPyC

unbounded computing

<http://rpyc.readthedocs.org/en/latest/>

- RPC system for Python
- Video demo: <http://showmedo.com/videotutorials/video?name=2780000;fromSeriesID=278>
- Classic and Service modes (we'll cover service)
- Any class method prefixed with 'exposed' is available remotely
- First class language/functions possible!

Distributed Object Frameworks

- Add state and OO to RPC
- Issues... Object activation & lifetime
 - Who activates an object?
 - Server (on startup)
 - Client (on connection)
 - When does an object die?
 - Server initiated - programmer decides
 - Client - ??? Depends!

Examples of Distributed object frameworks

- Common object request broker architecture (CORBA)
 - cross-platform, cross-language
 - complex, not very popular
- RPyC for Python, platform independent: XML-RPC, JSON-RPC
- [Thrift](#) (Facebook) - cross-platform RPC
- [Protocol Buffers](#) (Google)

Examples of Distributed object frameworks (cont)

- DCOM/COM+
 - Microsoft technology before .NET
 - Losing popularity quickly to .NET
- Java remote method invocation (RMI)
 - *stub-skeleton* concept: define (and publish) interface extending *java.rmi.Remote*
- .NET Remoting
 - Very flexible RPC (can specify transport system, marshaling method, lifetime, etc)
 - Uses TCP or *HTTP*

Service-oriented Computing

- ***Boundaries are explicit*** - remote interaction explicit - minimal interface
- ***Services are autonomous*** - general/used anywhere, not for specific thing - user must deal with errors
- ***Services share schemas and contracts, not class or interface definitions*** - not bound to language - common schemas (JSON, XML)
- ***Service compatibility is determined based on policy*** - structural compatibility and semantic compatibility

Abstract away specific implementation technology!

Service-Oriented Architecture (SOA)

Software system designed as a collection of interacting services.

2 Roles:

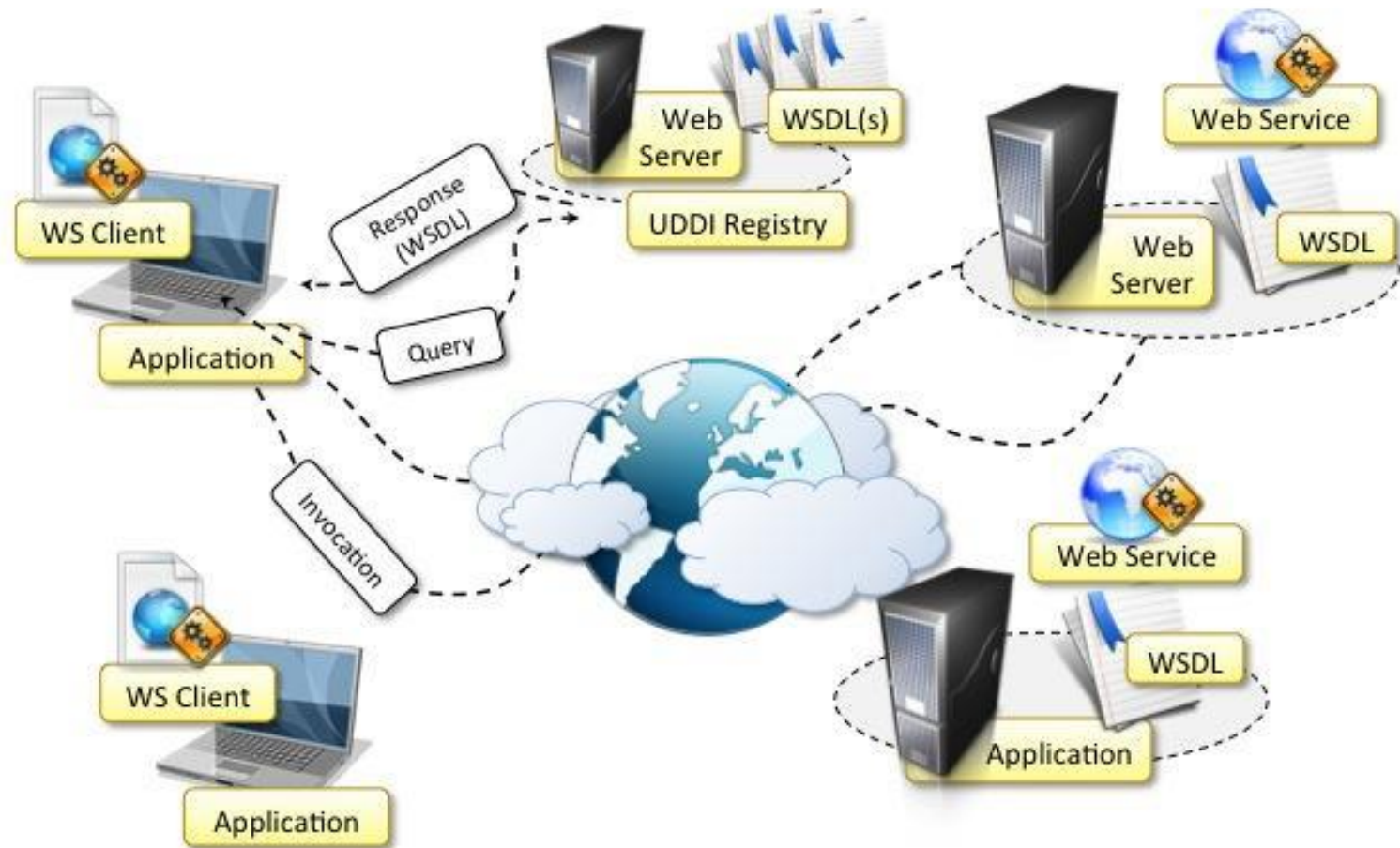
- Service provider
- Service consumer

SOA Enterprise Features

- Standardized service contract
- Loose coupling - minimize dependencies
- Abstraction - hide logic
- Reusability
- Autonomy
- Lack of state - easier to use
- Discoverability
- Composability

Web Services

SOA using HTTP, SOAP, XML, WSDL



Simple Object Access Protocol (SOAP)

XML language for exchanging information

```
POST /InStock HTTP/1.1
Host: www.stocks.com
Content-Type: application/soap+xml; charset=utf-8
Content-Length: <Size>
```

```
<?xml version="1.0">
```

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
```

Envelope

```
<soap:Header></soap:Header>
```

Header: Metadata & Assertions

```
<soap:Body xmlns:m=http://www.stocks.org/stock>
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>
```

Body: Method Call

```
</soap:Envelope>
```

Representational State Transfer (REST)

- Rely on HTTP PUT, GET, POST, DELETE for 'action' rather than embedding it in XML
- Can use XML or JSON for encoding
- *Web Service Description Language (WSDL) - document for describing web services*
 - *Designed for SOAP, though falling out of favor - we use API documentation (manual) or libraries*

Summary

- Parallel vs distributed computing
- Parallel architectures
- IPC, RPC, Distributed Objects, Active Objects, web services
- Client/Server, Peer-to-peer
- Point-to-point, publish-and-subscribe
- SOA, SOAP, REST, XML, JSON