Case for Distributed Systems

• Compare 80’s and now with respect to
  – Growth in processing power
    • Groch’s Law (back then: double price, get 4x computing power; today: maybe 50% = ½)
  – Network speed/capacity: computing on bytes used to be cheaper than shipping bytes – now bandwidth is “free”
  – Changes in software design from big systems to small connected components
• Conclusion: → Distributed Systems!

Definition

• Collection of independent computers that appears as a single coherent system.
  (Tanenbaum)
• Connects users and resources; allows users to share resources in a controlled way
• Lamport’s alternative definition:
  → You know you have one if the crash of a computer you’ve never heard of prevents you from getting any work done.

Examples of DS

• Client-Server systems
• Peer-to-Peer systems
  – Unstructured (e.g., exporting Windows shares)
  – Structured (e.g., Kazaa, Bittorrent, Chord)
• Clusters
• Middleware-based systems
• “True” distributed systems

Aside: Clusters

• Term can have different meanings:

(n.) A group of computers connected by a high-speed network that work together as if they were one machine with multiple CPUs.
docs.sun.com/doc/805-4368/8i450o0o

A group of independent computer systems known as nodes or hosts, that work together as a single system to ensure that mission-critical applications and resources remain available to clients.
www.microsoft.com/...serv/reskit/distsys/dsggloss.asp
Dist’d Systems vs OS

- Centralized/Single-processor Operating System
  - manages local resources
- Network Operating Systems (NOS)
  - (loosely-coupled/heterogeneous, provides interoperability)
- Distributed Operating Systems (DOS)
  - (tightly-coupled, provides transparency)
    - Multiprocessor OS: homogeneous
    - Multicomputer OS: heterogeneous
- Middleware-based Distributed Systems
  - Heterogeneous + transparency

Network OS

- Distributed system built on top of computers running a NOS: nodes export local services

Client/Server Paradigm

- Request/reply behavior: typical for NOS

Distributed Operating System

- Can be multiprocessor or multicomputer

“True” Distributed System

- Aka single system image
- OS manages heterogeneous collection of machines
  - Integrated process + resource management
  - Service migration
  - Automatic service replication
  - Concurrency control for resources built-in
- (Partial) examples: Apollo/Domain, V, Orca

Middleware-based System

- Typical configuration:
  - Distributed system built on middleware layer running on top of NOS
Middleware-based System (cont’d)

- Aka “three-tiered” architecture
  - Application/Middleware/OS
- Focus on
  - High-level communication facilities – paradigms
    - Distributed objects, distributed documents
  - Distribution transparency through naming
  - Persistence (file system/database)
  - Distributed transactions
  - Security
- Typically no management of local resources

Three-tiered Architecture

- Observation: server may act as client

Goals for Distributed Systems

- Transparency
- Consistency
- Scalability
- Robustness
- Openness
- Flexibility

Kinds of Transparency

<table>
<thead>
<tr>
<th>Transparency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and how a resource is accessed</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where a resource is located</td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that a resource may move to another location</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that a resource may be moved to another location while in use</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that a resource may be replicated</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that a resource may be shared by several competitive users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide the failure and recovery of a resource</td>
</tr>
<tr>
<td>Persistence</td>
<td>Hide whether a (software) resource is in memory or on disk</td>
</tr>
</tbody>
</table>

Degrees of Transparency

- Is full transparency always desired?
- Can’t hide certain physical limitations, e.g. latency
- Trade-off w/ performance
  - Cost of consistency
  - Consider additional work needed to provide full transparency

Consistency

- Distributed synchronization
  - Time
    - Logical vs physical
  - Synchronization of resources
    - Mutual exclusion, election algorithms
- Distributed transactions
  - Achieve transaction semantics (ACID) in a distributed system
- Example of trade-off:
  - Consistency in a distributed shared memory system
  - Global address space: access to an address causes pages containing the address to be transferred to accessing computer
False Sharing

Types of Scalability

- Size
  - Can add more users + resources
- Geography
  - Users & resources are geographically far apart
- Administration
  - System can span across multiple administrative domains

Q.: What causes poor scalability?

Centralization Pitfalls

- Centralized services
  - Single point of failure
- Centralized data
  - Bottlenecks → high latency
- Centralized algorithms
  - Requiring global knowledge

Decentralized Algorithms

- Incomplete information about global state
- Decide based on local state
- Tolerate failure of individual nodes
- No global, physical clock
- Decentralized algorithms are preferred in distributed systems, but some algorithms can benefit from synchronized clocks (e.g., leases)

» Brief review of clocks to follow

Discuss Clocks Next Tuesday

Skip the next slides

Clocks in Distributed Systems

- Physical vs Logical Clocks
- Physical:
  - All nodes agree on a clock and use that clock to decide on order of events
- Logical clocks:
  - A distributed algorithm nodes can use to decide on the order of events
Cristian’s Algorithm

Both $T_0$ and $T_1$ are measured with the same clock

Client
Request
Central server keeps time, clients ask for time
Attempts to compensate for latency – component of modern NTP Protocol – accuracy 1-50ms

Berkeley Algorithm

No time reference necessary

Logical Clocks Recap

• Lamport clocks are consistent, but they do not capture causality:
  – Consistent: $a \rightarrow b \Rightarrow C(a) < C(b)$
  – But not: $C(a) < C(b) \Rightarrow a \rightarrow b$
  
  (i.e., they are not strongly consistent)

• Two independent ways to extend them:
  – By creating total order (but not strongly consistent!)
    • $(C_i, P_m) < (C_k, P_n)$ if $C_i < C_k || (C_i == C_k && m < n)$
  – By creating a strongly consistent clock (but not a total order!)
    • Vector clocks

Vector Clocks

• Vector timestamps:
  – Each node keeps track of logical time of other nodes (as far as it’s seen messages from them) in $V_i[j]$
  – Send vector timestamp $vt$ along with each message
  – Reconcile vectors timestamp with own vectors upon receipt using $\text{MAX}(vt[k], V_i[k])$ for all $k$

• Can implement “causal message delivery”

Vector Clocks (1)

Vector Clocks (2)
Vector Clocks: Strong Consistency

- Definition:
  - \( V(a) < V(b) \):
    - \( V(a) \leq V(b) \) and there exists an \( i \) : \( V_i(a) < V_i(b) \)
    - \( V(a) \leq V(b) \): for all components \( i \) : \( V_i(a) \leq V_i(b) \)
- Strongly consistent:
  - \( a \rightarrow b \iff V(a) < V(b) \)
- Also:
  - \( a \parallel b \iff V(a) \parallel V(b) \)
  - \( \neg (V(a) < V(b) \vee V(b) < V(a)) \)

Applications of Logical Clocks

- Distributed mutual exclusion
  - Lamport’s algorithm
- Totally ordered multicast
  - For updating replicas
- Causal message delivery
  - E.g., deliver message before its reply
  - message or application layer implementation
- Distributed Simulation

Fault Tolerance

Fault Tolerance Terminology

- Distributed systems should tolerate partial failure
- Faults cause errors that lead to failure
- Aspects of dependability:
  - Availability
  - Reliability
  - Safety
  - Security
  - Maintainability
- Types of faults:
  - Transient vs. intermittent vs. permanent

Robustness/Fault Tolerance

- In Client/server communication:
  - Retransmission of requests, RPC semantics
- Distributed protocols
  - Byzantine Generals Problem
- Software Fault Tolerance
- Resource control
  - E.g., DDoS

Failure Models

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server fails, but is working correctly until it halts</td>
</tr>
<tr>
<td></td>
<td>Fail-Stop: if this can be detected</td>
</tr>
<tr>
<td></td>
<td>Fail-Silent: if not (and others may wrongly believe in crash)</td>
</tr>
<tr>
<td>Omission failure</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to send messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server’s response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td>The server’s response is incorrect</td>
</tr>
<tr>
<td>Value failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
<tr>
<td>(Byzantine failures)</td>
<td></td>
</tr>
</tbody>
</table>
Types of Redundancy

- Redundancy masks failures
- Information Redundancy
  - Example: Forward Error Correction (FEC)
- Time Redundancy
  - Example: Transactions
- Physical Redundancy
  - Example: Triple Modular Redundancy
- Let’s look at redundancy through replication

Types of Replication Protocols

- Primary-based protocols
  - hierarchical group
  - generally primary-backup protocols
  - options for primary:
    - may propagate notifications, operations, or state
- Replicated-write protocols
  - flat group using distributed consensus

Election Algorithms

- For primary-based protocols, elections are necessary
- Need to select one “coordinator” process/node from several candidates
  - All processes should agree
- Use logical numbering of nodes
  - Coordinator is the one with highest number
- Two brief examples:
  - Bully algorithm
  - Ring algorithm

Bully Algorithm (1)

Bully Algorithm (2)
**Ring Algorithm**

```
0 1 2 3 4 5 6 7

Previous coordinator has crashed
No response

Election message

[5,6,0] [5,6,1] [5,6,2] [5,6,3] [5,6,4] [5,6,5] [5,6,6]
```

**K fault tolerance (in Repl. Write)**

- A system is *k-fault tolerant* if it can tolerate faults in k components and still function
- Many results known; strongly dependent on assumptions
  - Whether message delivery is reliable
  - Whether there is a bound on msg delivery time
  - Whether processes can crash and in what way:
    - Fail-stop, silent-fail or Byzantine

**Reliable Communication**

- Critical: without it, no agreement is possible
- Aka “Two Army Problem”
  - Two armies must agree on a plan to attack, but don’t have reliable messengers to inform each other
  - Assumes that armies are loyal (=processes don’t crash)
- Hence all consensus is only as reliable as protocol used to ensure reliable communication
- Aside: in networking, causes corner case on TCP close

**Byzantine Agreement in Asynchronous Systems**

- *Asynchronous*:
  - Assumes reliable message transport, but with possibly unbounded delay
- Agreement is impossible with even one faulty process. Why?
  - Proof: Fischer/Lynch/Paterson 1985
  - Decision protocol must depend on single message:
    - Delayed: wait for it indefinitely
    - Or declare sender dead: may get wrong result

**Alternatives**

- Sacrifice safety, guarantee liveness
  - Might fail, but will always reply within bounds
- Guarantee liveness probabilistically
  - “Probabilistic Consensus”
  - Allow for protocols that may never terminate; but this would happen w/ probability zero
    - E.g., always terminates, but impossible to say when in the face of delay

**K-Fault Tolerance Conditions**

- To provide K-fault tolerance for 1 client with replicated servers
  - Need k+1 replicas if at most k can fail silent
  - Need 2k+1 replicas if up to k byzantine failures and all respond (or fail stop)
  - Need 3k+1 replicas if up to k byzantine failures and replies can be delayed
    - Optimal: both necessary and sufficient
Virtual Synchrony

- Recall: for replication to make sense, all replicas must perform the same set of operations
  - Generalization: "replicated state-machines"
- Virtual Synchrony:
  - Definition: A message is either delivered to all processes in a group or to none
- Keyword: delivered (vs. received)

Message Receipt vs. Delivery

Virtual Synchrony (cont’d)

- Observation: ensuring virtual synchrony requires agreement on group membership
- Group membership can change:
  - Processes may leave/crash
  - Processes may join/restart
- Idea: associate message m with current Group View (set of receivers)

Virtual Synchrony & Views

Atomic Multicast

= Virtual Synchrony + Total-order Delivery
Classification of virtual synchronous reliable multicasting:

<table>
<thead>
<tr>
<th>Message Ordering</th>
<th>None</th>
<th>FIFO Ordering</th>
<th>Causal Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not total-ordered</td>
<td>Reliable Multicast</td>
<td>FIFO Multicast</td>
<td>Causal Multicast</td>
</tr>
<tr>
<td>Total-ordered</td>
<td>Atomic Multicast</td>
<td>FIFO Atomic Multicast</td>
<td>Causal Atomic Multicast</td>
</tr>
</tbody>
</table>

Summary Fault Tolerance

- Fault tolerance in replicated write systems requires:
  - Distributed consensus
  - Which assumes atomic multicast, which must solve two subproblems
    - Virtually synchronous: same set of msg
    - Totally ordered: same order
Continuing on Scalability

- Recall: main problem that limited scalability was centralization (in services, in data, in centralized algorithms)
- Aside from using decentralized algorithms (where possible), what else can be done to increase scalability?

Scaling Techniques

- Hide communication latencies
  - Use asynchronous communication whenever possible

![Diagram showing synchronous vs. asynchronous communication]

Deferred synchronous RPC

- Combines two asynchronous RPC.

![Diagram of deferred synchronous RPC]

Scaling Techniques (cont’d)

- Minimize communication
  - Through distribution
  - Through piggybacking
  - Through careful placement of computation
  - Examples of these?
- Note shift in focus over time
  - as bandwidth becomes cheaper stronger focus on avoiding relative latency penalty

Latency lags Bandwidth

- Patterson [2004]
- Answers:
  - Caching
  - Replication
  - Prediction

Workload & Data Distribution

- DNS Zones

![Diagram of DNS zones and examples of zone names]
**Consistency Models**

- Scalability goal when using caching/replication:
  - minimize synchronization requirements
  - use relaxed consistency models when possible
- Consistency Models
  - Strict consistency
  - Sequential consistency; linearizability
  - Causal consistency
  - FIFO consistency
  - Weak consistency
  - Refinements: Release consistency, Entry consistency

**Strict Consistency**

- Any read on a data item x returns the value most recently written to x.
- Ideal model for programmers
  - Requires global clock (example: leases)

**Sequential Consistency**

- The result of the execution is the same as if reads and writes were executed in some sequential order; reads and writes of each process are executed in program order within that sequence

**Sequential Consistency (cont’d)**

- Note that sequential consistency requires
  - Maintaining constraints by program order
  - Data coherence within global sequence (“history”)
- Updates must be synchronous
  - Write update vs. write invalidate
- Performance: it has been shown that \( r + w > t \) where \( r \): read time, \( w \): write time, \( t \): message time
  - Optimizing writes makes reads slower & vice versa

**Causal Consistency**

- Not all processes need to see all writes in the same order
  - Causal consistency – only if writes are causally related (as in happens before relship)

**Causal Consistency (II)**

- Example of a violation: \( W(x)a \) happens before \( W(x)b \), so \( P3 \) and \( P4 \) must see results in same order
**Weaker Consistency Models**
- Idea: don’t propagate all updates, only propagate consistent state between updates to distributed synchronization variables
- Provide sequential consistency, but only with respect to sync points

**Release Consistency**
- Propagate writes when releasing a distributed synchronization variable
- Can be done eagerly or lazily
- Also possible: entry consistency
  - Only update those that will be accessed after entry

**Layered Architectures**
- Layer $k$ may interact with peer layer $k$ only via protocols
  - Layer $k+1$ interacts with layer $k$ via interface

**Layering and the E2E Argument**
- In any system using layering, designer has a choice of where to place functionality
  - Unless design by committee

**End-to-end argument**
- If correct & complete implementation requires help & knowledge only endpoints have, do not push the functionality down into lower layers
- Corollary:
  - A layer should only implement functionality that is needed by all clients, and can be completely implemented within that layer.

**E2E Examples**
- Careful file transfer
- Security & Encryption
- Error detection & correction
- Causal message delivery
E2E (cont’d)

• Note that endpoint != application
  – Endpoint can also be a layer
  – How to identify the endpoints?
• Reasons for violating E2E:
  – Performance
  – Cost
  – Software engineering/Code Reuse (?)
• E2E is only a guiding principle, a type of “Occam’s Razor”

Summary

• Transparency goal
• Techniques for scalability
• Consistency models
• Fault tolerance approaches & results