Distributed System Architectures

Distributed Systems

A distributed system is a system that
- consists of several machines that do not share memory or clock
- communicate via messages over a network
- have their own memory and other resources
- run their own OS

- different approaches in managing local and remote resources due to their different costs of using and managing them
Why Distributed Systems?

- Significant price/performance over centralized time-sharing systems
  - resource sharing
  - enhanced performance
  - improved reliability and availability
  - modular expandability
- Architecture types
  - minicomputer (#cpus / #users << 1)
  - workstation (#cpus / #users ~= 1)
  - processor pool (#cpus / #users >> 1)
- Distributed OS offer transparency in using system resources

Main Issues in Distributed Systems

- Global knowledge of system state
- Naming and name resolution of system resources
- Scalability
- Compatibility
- Process synchronization
- Resource management
  - data & computation migration
  - distributed scheduling
- Security
- Structure of system S/W
Global Knowledge & Synchronization

- Complete and accurate knowledge of all processes and resources is not easily available
- difficulties arise due to
  - absence of global shared memory
  - absence of global clock
  - unpredictable message delays
- challenges
  - decentralized system wide control
  - total temporal ordering of system events
  - process synchronization (mutex, deadlocks, starvation)

Naming

- Name/directory service maps logical names of system objects to physical addresses of those objects
- replicated directories
  - increase availability, reliability, and performance
  - replica maintenance and storage needed
- partitioned directories can overcome some of the problems with replicated directories but
  - finding relevant partition is still an issue
Scalability & Interoperability

- System performance should degrade gracefully, if at all, as the system grows

- interoperability levels of system resources
  - binary level (same binaries)
  - execution level (same source code)
  - protocol level (same set of protocols)

Resource Management & Security

- Resource management
  - make local & remote resources available to users transparently and in an effective and efficient manner
  - approaches
    - data migration (distributed file systems and shared memory)
    - computation migration (e.g. RPC)
    - distributed scheduling

- Security
  - authentication
  - authorization
Structure of Distributed Operating Systems

- **monolithic kernel**
  - one size fits all (diskless workstations, multiprocessors, and file servers)

- **collective kernel**
  - collection of largely independent processes providing different system services
  - micro-kernel provides essential services needed by all

- **object-oriented**
  - system services are implemented as collection of objects

- **client-server**

Introduction to Communication Networks

- **WANs, MANs, and LANs**
  - Consist of computers, links, switches, routers, bridges
    - use the store-and-forward technique
    - usage modes: packet switching vs circuit switching

- **ISO/OSI Reference model**
  - 7 layers: physical, datalink (transmission error recovery & flow control), network (routing & congestion control), transport, session, presentation, and application layers

- **LAN topologies**
  - bus and tree topologies
    - using the CSMA/CD and token bus protocols
  - ring topologies
    - using the token ring and slotted ring protocols
Communication Primitives

- SEND & RECEIVE message primitives
- Buffered vs un-buffered options for message primitives
  - Message copy path: source user buffer => source kernel buffer => destination kernel buffer => destination user buffer
- Types of communication primitives
  - Blocking vs non-blocking primitives
  - Reliable or unreliable primitives
  - Synchronous vs asynchronous primitives

Communication Primitives

- Difficulties with SEND & RECEIVE message primitives
  - Pairing responses with requests
  - Data representation
  - Addressing and failure recovery
  - Debugging & correctness
**Remote Procedure Call (RPC) Mechanism**

- Processes can communicate with remote processes via a procedure call and the help of
  - Stub procedure code (for both local and remote procedures) that
    - Local stub queries a binding server for server address of remote procedure, packs the procedure parameters, and then sends them to remote stub procedure
    - Remote stub procedure unpacks parameters, performs local procedure call, and then returns results to local stub,
    - Local stub unpacks the results and returns them to the calling local process
  - A binding server resolves procedure names to server addresses
- Core issues
  - Structure of RPC and binding
  - Parameter and result passing (by value or by reference)
  - Handling of failures – RPC semantics

**RPC Semantics**

- RPC calls can fail due to computer or communication failures
  - Semantics of an RPC call $P()$

<table>
<thead>
<tr>
<th></th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>#exec</td>
<td>Success</td>
<td>Failure</td>
</tr>
<tr>
<td>At least once</td>
<td>$\geq 1$</td>
<td>0 or more</td>
</tr>
<tr>
<td>Exactly once</td>
<td>1</td>
<td>0 or 1</td>
</tr>
<tr>
<td>At most once</td>
<td>1</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

- Correctness conditions: $C_1 \rightarrow C_2 \Rightarrow W_1 \rightarrow W_2$ where $C_i$ denote RPC calls and $W_i$ work done on shared data, and $\Rightarrow$ denotes a “happened before” relation
**RPC Issues**

- **Implementation issues for the RPC mechanism**
  - low latency RPC calls (use UDP)
  - high-throughput RPC calls (use TCP)
  - increase concurrency of RPC calls via
    - Multi-RPC
    - Parallel RPC
    - asynchronous calls

- **Shortcomings of the RPC mechanism**
  - does not allow for returning incremental results
  - remote procedures are not *first-class citizens* (e.g. can not be used everywhere where local procedures/variables can be used)